Modular Platform for Commercial Mobile Robots

Kjærgaard, Morten

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

Technical University of Denmark

Morten Kjærgaard

Modular Platform for Commercial Mobile Robots

PhD thesis, June 2013

DTU Electrical Engineering
Department of Electrical Engineering

Preface

This thesis is written as conclusion of my PhD project at the Technical University of Denmark, Department of Electrical Engineering. The project period was from May 2010 to May 2013, including a six month external stay at Willow Garage Inc. located in Menlo Park, California.

The project was carried out as an Industrial PhD through the The Industrial PhD Program from the Danish Ministry of Science, Innovation and Higher Education. The company sponsoring the project was Prevas A/S.

The supervisors of the project at the Technical University of Denmark were Ole Ravn and Nils Axel Andersen, and the supervisor at Prevas A/S was Jakob Koed during the first year and Peter Aagaard Kristensen during the last part of the project.

The main topic of the project was to understand the reasons why the recent advances in robotic technology did not result on more commercial products. The thesis addresses it in three ares. A survey of the existing technologies used in robotics to understand the commercial potential. A practical approach towards running the robot controllers on low cost and low powered embedded hardware. And a theoretical and model driven approach towards configuring and calibrating robot applications to gain better reliability and faster bring-up time of new products.

Morten Kjærgaard
June 2013

Summary

Despite a rapid development in computers and sensor technologies, surprisingly few autonomous robot systems have successfully made it to the consumer market and into people's homes. Robotics is a popular topic in research circles, but focus is often on ground-breaking technologies, and not on putting the robots on the commercial market.

At the time when this research project was started in May 2010, the amount of successful commercial applications based on mobile robots was very limited. The most known applications were vacuum cleaners, lawn mowers, and few examples of specialized transport robots used in warehouses and hospitals.

At the same time, despite attempting to solve the same tasks and applications, the resulting software and products of research groups was very fragmented. Even if being open source, the software was based on self-made frameworks and often only used internally by the individual groups and perhaps a few close industrial partners.

This research project addresses the problem of increasing the potential for more commercial applications based on mobile wheeled robots. Therefore the main focus is not on inventing new ground-breaking robotics technology, but instead understanding why the existing technology and algorithms are not ready for production.

One focus area of this project is an analysis of these existing technologies and algorithms for mobile robots. The most fundamental task for a mobile robot is navigation, yet no generic and ready to use implementation for solving this exists. This project includes an effort towards such

a generic navigation system. It should provide a stable and easy-to-set-up experience for robotics researcher and industry integrators who needs navigation capability for a specific mobile robot. At the same time a common package for navigation will provide a base for many researchers to contribute to and mature over time.

The second focus area was to close the gap between research and industry by providing the necessary tools and motivation for researchers to create more robust prototype applications. During the time of the project period, a significant research community was created around one specific robot control framework called ROS. From the very beginning, this research project acknowledged the value of such a community, and put a significant effort into influencing the ROS framework to become usable also for industry and commercial applications. Based on a requirement analysis for such a framework, a prototype implementation of an industry ready component based ROS compatible middleware was created.

The project also includes work towards a smart parameter framework, assisting in configuring the individual components in a component based control framework. The smart parameters adapt to the respective robot, and makes it possible to reuse advanced software components, without expert knowledge about the underlying algorithms. The smart parameters also assists in building a robot system, that can autonomously calibrate and optimize itself.

Resume

På trods af en hastig udvikling indenfor computere og sensor-teknologier, findes der overraskende få autonome robot-systemer som kommercielle produkter. Robotteknologi er et populært forskningsemne, men der fokuseres ofte på nye banebrydende teknologier og ikke på at få robotterne på markedet.

Da dette projekt blev påbegyndt i maj 2010 var det meget begrænset hvad der fandtes a kommercielle autonome robotsystemer. De mest udbredte var støvsuger-robotter, græsslåmaskiner og specialdesignede transport-robotter til hospitaler og lagre.

I samme periode var det meget begrænset hvad der blev delt af software implementatationer mellem de forskellige forsknings-grupper. Selvom funktionalitet blev implementeret som open-source, byggede det ofte på hjemmelavede frameworks, hvilket gjorde det svært at benytte for andre.

Dette forskningsprojekt arbejder med at forbedre potentialet for at skabe flere kommercielle produkter baseret på autonome mobile robotter. Hovedfokus er derfor ikke at opfinde ny banebrydende teknologi, men at se på den eksisterende teknologi og fjerne de forhindringer der er for at den benyttes i flere kommercielle produkter.

Et af fokusområderne er en analyse af eksisterende teknologier og algoritmer der benyttes til mobile robotter. På trods af at navigationen er den mest grundlæggende opgave for sådan en robot, findes der ikke en generisk og robust implementation der kan løse denne opgave. Dette projekt arbejder hen imod sådan en implementation af et generisk

navigations-system. Formålet er at gøre navigation til en let tilgængelig egenskab for en vilkårlig robot, samt have en fælles implementation som kan modnes og udvikles i fællesskab.

Et andet fokusområde var at mindske hullet mellem forskning og industri, ved at skabe motivation og levere værktøjer der gør forskere bedre i stand til at udvikle deres algoritmer som robuste og genbrugelige moduler. I løbet af projektperioden opstod der et betydeligt community omkring ét specifik framework til robot control systemer kaldet ROS. Værdien af sådan et community var tydelig set fra dette projekt, og forsøgte derfor fra starten at påvirke ROS framework'et til at kunne benyttes til industrielle og kommercielle produkter. Som resultat deraf, og efter en større krav-analyse, blev der udviklet en prototype af et forbedret ROS kompatibelt komponent baseret control framework som kunne benyttes industrielt.

I projektet blev der også arbejdet på en model-baseret konfiguration af disse komponenter. Ved at benytte såkaldte "Smart Parameters" der ud fra en model kan tilpasse sig til den pågældende robot, bliver det meget simplere at benytte avancerede software komponenter, da man ikke behøver at kende til detaljerne om hvordan algoritmerne fungerer og konfigureres. Vha. disse parametre og tilhørende model, er det også muligt få robotten til at starte med ukendte parametre og selv kalibrere og løbende justere sig selv.

Acknowledgement

First of all, I would like to thank the management team and my company supervisors at Prevas A/S Denmark for giving me the opportunity to carry out this PhD project. Special thanks to Rune Domsten, Jakob Koed and Peter Aagaard Kristensen for the support and inspiring discussion to help define and guide the project from the start to the very end.

I would also like to thank my university supervisors Ole Ravn and Nils Axel Andersen for the support and guidance, and for welcoming me into the department. They always have an open door, a helpful attitude, and their experience and knowledge about the robotics research has been a great help during the project.

Also many thanks to my colleges and the other interns during my external stay at Willow Garage Inc. Special thanks to my supervisor and mentor at Willow Garage, Troy Straszheim, an exceptionally inspiring person and software developer, from whom I have learned much more than I would have thought would be possible.

Thanks to my colleges at both the Technical University of Denmark and at Prevas A/S for making me feel welcome despite having to alternate between so many different work places.

The greatest thanks are to Ida and Sigurd, for their love and support, especially in the last hectic months of the project.

0. ACKNOWLEDGEMENT

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

1.1 Background

Robotics is a popular and challenging research topic at many universities around the world. Often you experience these research groups showing advanced robot applications capable of handling some specific work-task. These projects are often based on a standard research platform modified to the specific purpose and the functionality implemented using a robotics software framework.

These numerous examples of advanced research robot applications show that it is technically possible to construct quite advanced task-handling robots, but the real deployment of robot applications in everyday life has however been quite limited to comparison. Moving and maturing a robot application from a working laboratory research prototype to a commercial real-life applications is challenging. Many new requirements arise, such as safety, cost efficiency, and proper reliability to run in longer periods without human assistance.

Prevas A/S is a development company who is often assisting in the development and implementation of novel and innovative products. Mobile and autonomous robot systems is an area where potential customers often bring in ideas and request for product development, but where both the risks and development costs turns out to be too high with the current

state of technology. To overcome this limitation in the future Prevas A/S has identified several main causes:

- The cost and risk associated with developing a specific autonomous robot application from scratch are too high to make the project feasible and profitable. One cause is the massive need for advanced technology still in the research phase, and the complicated integration associated with such technology.
- Commercial robot applications deployed outside of an controlled laboratory environment are subject to a high level of uncertainty. Despite this, it should be capable of performing its task and run stable for long periods without local human assistance. High downtime or frequent need for technical on-site assistance is expensive and gives a bad impression of the robot.

1.2 Hypothesis

The project is based on the following hypotheses:

- It will be possible to structure the implementation of an autonomous mobile robot in a modular way such that: (1) Technology and implementation can be shared between projects in both research and industry. (2) The individual modules can be implemented, tested and matured in an isolated way.
- It will be possible to create a parameterized model for an autonomous mobile robot describing the physical configuration, including kinematics, dynamics, sensors, and actuators.
- It will be possible to create a parameterized model for an autonomous mobile robot describing the required behavior, including actions, low level tasks, and high level tasks.
- It will be possible to initially run the robot system with unknown or estimated parameter values for these models, and these parameters

can be adjusted autonomously by the robot by: (1) An initial self-calibration when the robot is deployed in its working environment. (2) An on-line self-optimization when the robot is performing its work-tasks.

1.3 Goals

Since the project is being carried out as an IndustrialPhD project the goals are specified for three areas. Development, scientific, and commercial.

1.3.1 Development Goals

The overall goal is to decrease the distance between that is technologically possible within research, and what is feasible and profitable to build as a commercial mobile robot applications by:

- Developing tools and methods that will assist in the development process of a robot application including: Reuse of implementation, faster prototyping, and testing with the purpose of carry out development projects with lower risk and cost.
- Contribute to, and influence the robotics research community, to motivate other researchers to make their work more robust and easier to use for commercial applications.

1.3.2 Scientific Goals

The goals are not to invent new ground-breaking technologies but instead:

- Analyzing the available technologies and algorithms used within the field of autonomous mobile robots, and their potential maturity for use in commercial products.
- Mature one or more technologies or algorithms for industrial use.

Additionally the project approaches a series of scientific problems concerning modeling, parameterization and self-calibration of a mobile robot. The project focuses on methods to implement the functionality of the robot application in a series of building blocks that are integrated with a parameterized model. The main topics are:

- Create a parameterized model for describing the physical properties and behavior of a robot.
- Self-calibration: Where the model-parameters are derived autonomously by the robot with minimal human interaction.
- Self-optimization: Where the model-parameters are optimized autonomously when the robot is performing its task.

1.3.3 Commercial Goals

The commercial goal of the project is to improve the capability of Prevas A/S to develop profitable autonomous robot applications, thus positioning the company as an attractive partner within field. Prevas A/S has previously had several request to develop high technology autonomous robots, where the customer is hesitating due to the high development cost and risk associated with such a product.

1.4 Contributions

The contributions covered by this thesis can be divided into the following three areas:

1. Robotics Middleware for Industrial and Embedded Systems

To be able to implement the robot control system in a modular and reusable way, it is required to use a robotics middleware. This thesis presents an analysis of available robotics middleware used in research. It also includes an in depth analysis of the additional requirements that

arises, when using such a middleware for commercial and industrial applications. As a conclusion, the thesis presents a next-generation decentralized middleware named "DARC", designed during the project. It is capable of fulfilling the industrial requirements, replacing the popular Robot Operating System (ROS) middleware, while still taking advantage of the large library of ROS functionality.

The design of the "DARC" middleware is also presented in the following publication:

- M. Kjaergaard, N. A. Andersen, O. Ravn "DARC: Next Generation Decentralized Control Framework for Robot Applications" In: *2013, 10th IEEE International Conference on Control and Automation (ICCA 2013)*

2. Generic Navigation for Mobile Wheeled Robots

This thesis presents several new methods towards a generally usable navigation package for wheeled mobile robots. The work includes a novel method to represent a trajectory based on fifth order Bézier curves, with support for rotation on the spot and reverse motion. It also presents a method to make the curvature continuous throughout the trajectory so it can be driven in a smooth motion. In addition, a method to create a velocity profile that complies with the robots dynamic constraints is presented.

Practical experiments with a differential drive robot is carried out using a custom designed control law. The custom control law is designed to make the robot capable of following both forward, reverse and rotation on the spot trajectory segments.

The work in this area is also presented in the following publication:

- M. Kjaergaard, N. A. Andersen, O. Ravn "Generic Trajectory Representation and Trajectory Following For Wheeled Robots" Submitted to: *2014 IEEE International Conference on Robotics and Automation (ICRA 2014)*

3. Parameterized Models for Configuring Robot Applications

This thesis presents a conceptual design towards a configuration framework, that can be used to configure the individual components in a component based robotics framework. The actual parameter values are extracted from easy to understand robot properties, defined in a robot specific hierarchical configuration model. In addition, the parameters include meta-information describing the origin of the value. With this information, the control system can start up partially with uncertain or even unknown parameter values, and perform calibration and optimization procedures. The purpose is to achieve more robust self calibrating robot application, and make it easier to reuse advanced robot algorithms without expert knowledge.

The work in this area is also presented in the following publication:

- M. Kjaergaard, N. A. Andersen, O. Ravn, P. A. Kristensen "Towards Competitive Commercial Autonomous Robots: The Configuration Problem" In: *2011, 16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA 2011)*

1.5 Contributions not covered in this thesis

The project also included work in the field of terrain mapping for outdoor robots. A probabilistic method was considered for extracting the terrain maps based on a point cloud measurement of the scene. The method uses Gaussian process regression to predict a estimate function and its relative uncertainty. The point cloud was extracted using a laser scanner and a 3D vision system. The work is covered by the following publication:

- M. Kjaergaard, A. S. Massaro, E. Bayramoglu, K. Jensen "Terrain Mapping and Obstacle Detection using Gaussian Processes" In: *2011, 10th International Conference on Machine Learning and Applications and Workshops (ICMLA'11)*

1.6 Outline

The rest of the thesis is structured as:

Chapter 2 presents the theoretical background for controlling wheeled mobile robots. This includes representations for pose, velocity and motion curvature. Special focus is put on Ackermann and differential drive robots, and includes a presentation of their geometrical properties and the constraints that must be considered.

Chapter 3 includes an example based survey of available mobile robot platforms and sensor technologies, with special focus on robustness, cost, and their application in consumer products.

Chapter 4 presents the work towards a generic navigation package based on the notion from chapter 2. The chapter includes a presentation of robot navigation in general and a survey of the different available methods. It presents a generic representation of a trajectory, a method to provide continuous curvature, a method to create a velocity profile and a controller for trajectory-following.

Chapter 5 presents an analysis of popular control middleware used for robotics applications in research. It also includes a requirements analysis, with the purpose of identifying the extra requirements that arise when a middleware is to be used for industrial and commercial applications.

Chapter 6 presents the design and functionality of a next generation middleware called "DARC". It is designed to be able to replace the ROS middleware and to satisfy both industrial and research requirements.

Chapter 7 presents a conceptual design towards a configuration framework, that can be used to configure the individual components in a component based robotics framework. It is called "Smart Parameter Framework" since it allows parameters to be smart and adapt to a new system provided a descriptive model of the robot.

Chapter 8 wraps up the work and contains the conclusion and future work.

2

Wheeled Robots

2.1 Introduction

Wheeled robots are a subcategory of mobile robots where motorized wheels in contact with the ground are used to drive the robot. They are popular in both research and in the consumer market because they are simple to control, and normally when equipped with a minimum of three wheels, they are stable even without any control input. As often simplicity comes with a price. Wheeled robots works best on a flat surface with proper friction, and usually are not fit for navigating over obstacles, on steep surfaces and soft and rocky terrain. Some of these limitations can be helped by equipping the robot with large wheels and powerful motors allowing the robot to navigate on fields and unstructured terrain, but this increases the cost and decreases the precision of the robot.

Wheeled robots can be designed with many different wheel combinations. Since the work in this project is focused towards consumer robots the scope has been limited to two of the most popular and cost efficient configurations.

1. Ackermann type robots with steerable front wheels and non-steerable rear wheels, as found in most modern cars.

2. Differential drive robots, with two individually powered wheels, and

a number of passive wheels for stability.

These two wheel combinations are somehow related because neither allows the robot to perform a sideways motion. The navigation strategy must take this into account.

2.1.1 Contribution

This chapter provides a formalization of some of the theory behind controlling wheeled robots with special focus on these two wheel configurations. This includes an analysis of the constraints the wheel configurations and other factors impede on the available velocity space for the robot. The purpose is to provide a background analysis used for the generic navigation in chapter 4.

Some wheel combinations does allow for full translational and rotational motions, so called omni-directional robots. Omni-directional robots can be built in several ways e.g. with Swedish wheels, or with exclusively steerable wheels. Because one would often use a different control strategy to take advantage of this capability, this type of robots are not covered here.

The chapter also shows how representing the velocity vector in polar coordinates makes it easier to represent rotation-on-the-spot motion, where no forward motion is taking place, and the curvature is infinite.

2.2 General Notation

2.2.1 Pose

This thesis adopts a notation for the robot state similar to [SNS11], changed slightly to fit better for both differential drive and Ackermann steered robots.

The robot is modeled as a rigid body moving on a flat surface. The robot frame $\mathbf{R}$ is orientated with the x-axis in forward direction, the y-axis to the left, and the z-axis upwards, thus positive rotation results in

counterclockwise rotation.

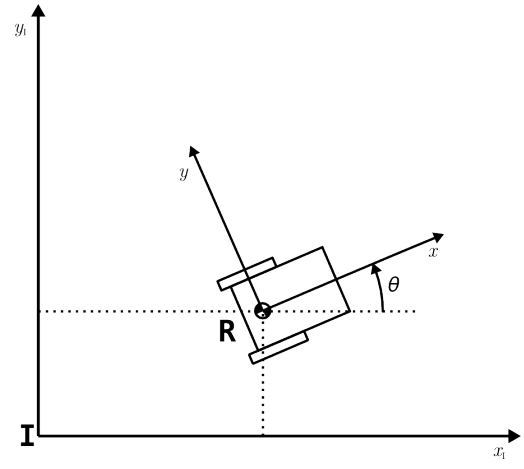


Figure 2.1: Frames and pose for a wheeled robot

The robot frame is defined by the pose of the robot, denoted ξ, which is expressed relative to an inertial reference frame **I**.

$$\xi = (x, y, \theta)^T \tag{2.1}$$

2.2.2 Velocity

The state of the robot futher includes the current velocity, expressed relative to the inertial frame as:

$$\dot{\xi}_I = (\dot{x}_I, \dot{y}_I, \dot{\theta}_I)^T \tag{2.2}$$

The velocity without a suffix is expressed relative to the robot. The velocity is transformed into the robot frame with the 3 dimensional rotation matrix.

$$\dot{\xi} = R(\theta)\,\dot{\xi}_I \tag{2.3}$$

$$= (\dot{x}, \dot{y}, \dot{\theta})^T \tag{2.4}$$

where

$$R(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

2.2.3 Curvature

The kinematics constraints of both the Ackermann and the differential drive robot does not allow for sideways motion. It is assumed that no skid is taking place, thus:

$$\dot{y} = 0 \tag{2.6}$$

Therefore the robot velocity can be expressed using only a forward velocity and an angular velocity $(\dot{x}, \dot{\theta})^T$. With such a velocity the robot will perform a curved motion corresponding to following a circle of radius r given by (2.7). Figure 2.2 shows an example of how a point **P** follows a circle of radius r when following a curve. The inverse of the circle radius is the curvature κ which can be found using (2.9).

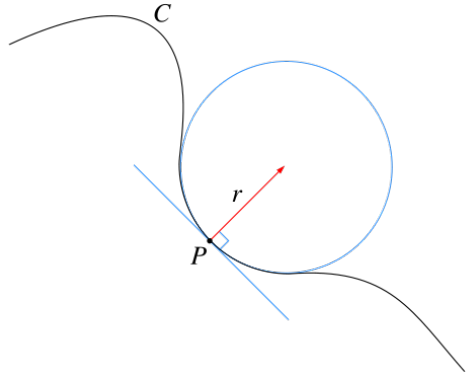


Figure 2.2: Example of how curvature is represented by a circle of radius r.

$$r = \frac{\dot{x}}{\dot{\theta}} \tag{2.7}$$

$$\kappa = \frac{1}{r} \tag{2.8}$$

$$= \frac{\dot{\theta}}{\dot{x}} \tag{2.9}$$

2.2.4 Polar Representation of Velocity Vector

As seen from (2.9), the curvature κ has the unfortunate property of becoming infinite when the robot is rotation on the spot and the forward velocity is zero. Rotation on the spot is a perfectly valid motion for a differential drive robot. To overcome this limitation, the velocity vector of $(\dot{x}, \dot{\theta})^T$ is instead represented in polar coordinates $(\rho, \phi)^T$, as illustrated in figure 2.3.

$$\rho = \sqrt{\dot{\theta}^2 + \dot{x}^2)} \tag{2.10}$$

$$\phi = \text{atan2}(\dot{\theta}, \dot{x}) \tag{2.11}$$

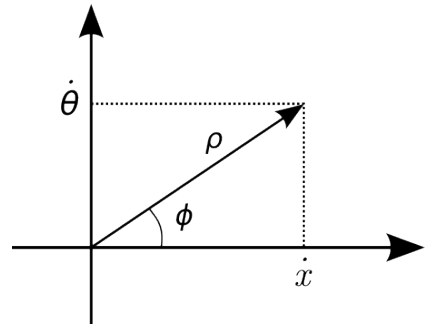


Figure 2.3: Polar representation of velocity vector

While the value of ρ is independent of the curvature of the robot motion, the value of ϕ relates directly to it with the following expression:

$$\kappa = \tan(\phi) \tag{2.12}$$

Thus $\phi = 0$ gives a straight motion, while $\phi = \frac{\pi}{2}$ and $\phi = -\frac{\pi}{2}$ is rotation on the spot. For ϕ the range $]-\frac{\pi}{2}, \frac{\pi}{2}[$ represents forward motion,

while $]-\pi, -\frac{\pi}{2}[$ and $]\frac{\pi}{2}, \pi[$ represents reversing. Planning with respect to curvature constraints can thus be planned taking only the ϕ factor of the velocity vector into account.

2.3 General Constraint

The motion of a wheeled robot is often constrained in several ways. Some constraints are geometric constraints resulting from the geometry of the robot, others are dynamic constraints limiting the velocity of the robot.

This section formalizes and investigates general constraints for all types of robots, while later sections will describe the robot specific constraints.

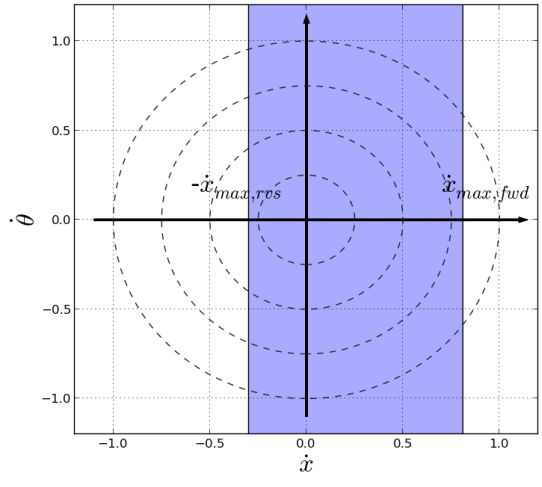


Figure 2.4: Example of how the driving velocity constraint affects the available velocity space. $\dot{x}_{max,fwd} = 0.8\frac{m}{s}$ and $\dot{x}_{max,rvs} = 0.3\frac{m}{s}$

2.3.1 Driving Velocity Constraint

The maximum velocity of the robot is usually limited due to factors such as safety or to controller characteristics. The optimal magnitude

can change dependent on the area the robot is driving in, the expected surroundings, and the observing capabilities of the sensors. Often the sensors are prioritized in the forward direction of the robot, so the maximum reversing velocity can be lower than the forward velocity. Assuming the maximum forward and reverse velocity of the robot is constrained by $\dot{x}_{max,fwd}$ and $\dot{x}_{max,rvs}$ respectively as defined in (2.13)-(2.15). Figure 2.4 illustrates how this constraint affects the available velocity space.

$$-\dot{x}_{max,rvs} \leq \dot{x} \leq \dot{x}_{max,fwd} \tag{2.13}$$

$$\dot{x}_{max,rvs} \geq 0 \tag{2.14}$$

$$\dot{x}_{max,fwd} \geq 0 \tag{2.15}$$

Given these limits, the maximum allowed magnitude of ρ, during a motion with a given curvature represented by ϕ can be found with the function in (2.16).

$$f_{\rho_{\max,\text{drive}}}(\phi) = \begin{cases} \frac{\dot{x}_{max,fwd}}{\cos\phi} & \text{when } \phi \in [-\frac{\pi}{2}; \frac{\pi}{2}] \\ \frac{\dot{x}_{max,rvs}}{\cos\phi} & \text{otherwise} \end{cases} \tag{2.16}$$

2.3.2 Centrifugal Force Constraint

When driving in a curve the robot will be exposed to a centrifugal force acting sideways on the robot. This force can make the wheels slide sideways, or even worse make the robot fall over or drop a load it is carrying.

The maximum centrifugal force the robot can handle depends on many factors such as the wheels, floor friction or the weight of the current load, and it is assumed that the maximum value has been determined. The centrifugal acceleration a_c acting on a robot with a given velocity vector is:

$$a_c = \frac{\dot{x}^2}{r} \tag{2.17}$$

$$= \kappa\, \dot{x}^2 \tag{2.18}$$

$$= \dot{x}\, \dot{\theta} \tag{2.19}$$

Figure 2.5: Centrifugal force making a car tilt in a curve

and similar when given a polar representation of the velocity vector:

$$a_c = \frac{1}{2}\sin(2\phi)\rho^2 \tag{2.20}$$

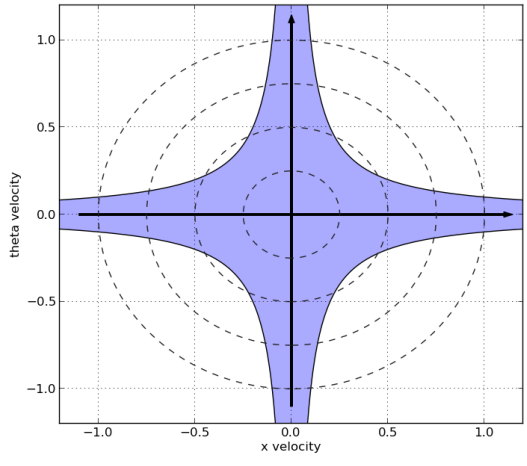


Figure 2.6: Example of how the centrifugal force constraint affects the available velocity space. $a_c = 0.1\frac{m}{s^2}$

Based on (2.20) the maximum allowed magnitude of ρ, during a motion with a given curvature represented by ϕ can be found with the function in (2.21).

$$f_{\rho_{max},a_c}(\phi) = \sqrt{\frac{2a_c}{\sin(2\phi)}} \tag{2.21}$$

The centrifugal force constraints the magnitude of both $\dot{x}$ and $\dot{\theta}$. Figure 2.6 illustrates how the centrifugal force constraint affects the available velocity space.

2.4 Ackermann Robots

Studies in wheeled robotics are often based on differential drive robots or omni-directional robots, since they have a much better maneuverability than Ackermann robots. In addition, the Ackermann steering is mechanically more complex to build.

Despite these limitations, the popularity of Ackermann steering in other areas such as car design does make the platform quite interesting to work with. It has a few noticeable advantages:

1. The configuration does not require any additional passive wheels to obtain stability
2. Better stability at high velocity
3. Only one powerful drive motor required

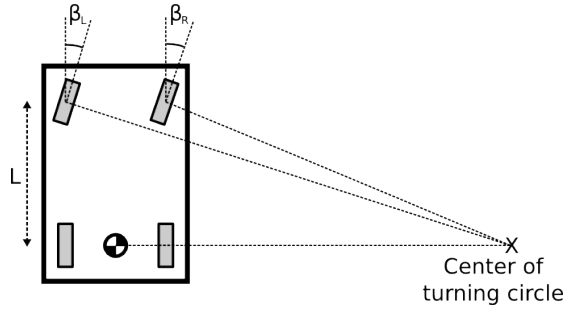


Figure 2.7: Ackermann Geometry

A standard Ackermann wheel configuration consist of two steerable front wheels and two powered rear wheels. To avoid making the front wheels slide sideways when turning, the two front wheels should have a slightly different steering angle. Figure 2.7 shows the outline and wheel positions of an Ackermann robot. The robot frame is traditionally placed between the two rear wheels. As the figure illustrates, the center of the circle of rotation is located in the intersection between the line from the rear-wheel axis, and the lines from the two steering wheel axes, which should all intersect in a single point. In addition, the two rear wheels should rotate with different velocity when turning. More details about Ackermann steering geometry, and its use in wheeled robots can be found in [DJ00].

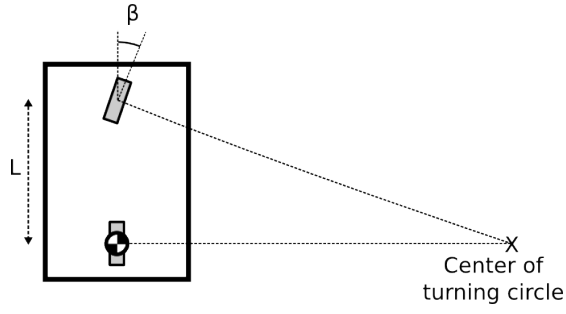


Figure 2.8: Unicycle Geometry

In practice the complexity of the Ackermann steering geometry is often handled mechanically or in the low level control. Therefore, for simplicity and without loss of generality, the Ackermann geometry can be modeled by the more simple bicycle-model illustrated in figure 2.8. In this case, the robot is assumed to be controlled by a single steering angle and a single rear wheel velocity.

2.4.1 Kinematic Model

The wheel configuration is parameterized by the length L between the front wheel and the rear wheel, and the radius r of the drive wheel. The

control state, denoted Φ, is the angular velocity of the drive wheel φ, and the steering angle β.

$$\Phi = (\varphi, \beta)^T \tag{2.22}$$

Given a control state and robot parameters, the curvature followed by the robot is:

$$\kappa = \frac{\tan(\beta)}{L} \tag{2.23}$$

The following equation relates the control state and the parameters to the velocity in the robot frame.

$$\dot{\xi} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} r\varphi \\ 0 \\ r\varphi \frac{\tan(\beta)}{L} \end{pmatrix} \tag{2.24}$$

And similar, the inverse equations relates a desired robot velocity to a control state.

$$\varphi = \frac{\dot{x}}{r} \tag{2.25}$$

$$\beta = \tan^{-1}\left(\frac{\dot{\theta}}{\dot{x}}L\right) \tag{2.26}$$

2.4.2 Steering Angle Constraint

Due to mechanical constraints an Ackermann wheel configuration will have a limited steering range defined as: $\beta \in [-\beta_{max}, \beta_{max}]$. The steering angle constraint results in a range of curvatures that the robot is incapable of following. It is independent on the driving speed of the robot, thus it can not be obeyed simply by limiting the driving speed of the robot. Instead it should be considered at the planning level, only creating trajectories with feasible curvatures.

As seen from (2.23) the range of possible curvatures, expressed by a allowed range for ϕ is given by (2.27) and (2.28) and illustrated in figure 2.9.

$$\phi_{\beta,max} = \tan^{-1}\left(\frac{\tan(\beta_{max})}{L}\right) \tag{2.27}$$

$$\phi \in [-\phi_{\beta,max}\,;\,\phi_{\beta,max}] \tag{2.28}$$

The constraint maps into limits for ρ using the following function:

$$f_{\rho_{\max},\text{steering}}(\phi) = \begin{cases} 0 & \text{when } \phi > \phi_{\beta,max} \\ \infty & \text{when } \phi \leq \phi_{\beta,max} \end{cases} \tag{2.29}$$

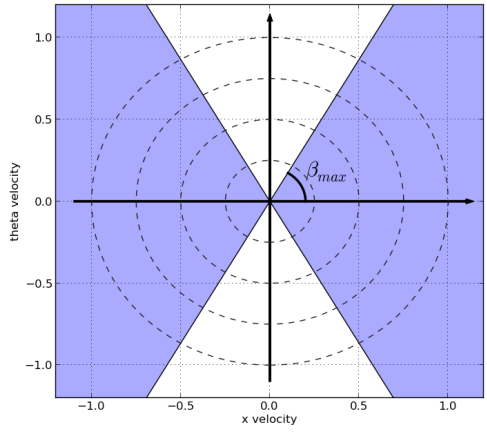


Figure 2.9: Example of how the steering angle constraint affects the available velocity space.

2.5 Differential Drive Robots

A standard differential drive robot has two separate drive wheels mounted on each side. The speed of each of the wheels can be controller individually making forward, reverse, curved, and rotation-on-the-spot motions possible. The center of rotation lies somewhere on the line extended through the drive wheel axis. With only two wheels the robot would have an unstable balance thus additional passive wheels such as caster wheels are added for stability.

Figure 2.10: The Adept MobileRobots Laser Powerbot is an example of a differential drive robot

Figure 2.11 shows a model of a differential drive robot, where the robot frame is defined in the middle between the drive wheels.

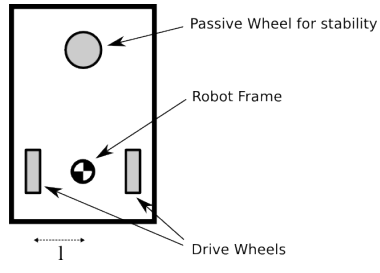


Figure 2.11: Differential Drive Geometry

2.5.1 Kinematic Model

The differential drive robot is parameterized by the radius r of the drive wheels and the length l from the robot frame and to each of the drive wheels. The control state Φ is the angular velocity of the left and right drive wheel respectively:

$$\Phi = (\varphi_r, \varphi_l)^T \tag{2.30}$$

The robot velocity, given robot parameters and a control state can be found with the following linear equation:

$$\mathbf{R} = \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.31}$$

$$\mathbf{O} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \\ \frac{1}{2l} & -\frac{1}{2l} & 0 \end{bmatrix} \tag{2.32}$$

$$\dot{\xi} = \mathbf{O}\,\mathbf{R} \begin{bmatrix} \Phi \\ 0 \end{bmatrix} \tag{2.33}$$

The inverse equation relates a desired robot velocity to a control state.

$$\mathbf{O^{-1}} = \begin{bmatrix} 1 & 0 & l \\ 1 & 0 & -l \\ 0 & 1 & 0 \end{bmatrix} \tag{2.34}$$

$$\begin{bmatrix} \Phi \\ 0 \end{bmatrix} = \mathbf{R}^{-1}\,\mathbf{O}^{-1}\,\dot{\xi} \tag{2.35}$$

2.5.2 Drive Wheel Constraint

Since each of the drive wheels are controlled individually it is possible that they each have a maximum allowed angular velocity, denoted φ_{max}.

$$-\varphi_{max} \leq \varphi_r \leq \varphi_{max} \tag{2.36}$$

$$-\varphi_{max} \leq \varphi_l \leq \varphi_{max} \tag{2.37}$$

Using the kinematics model in (2.33), the drive wheel constraint limits the magnitude of ρ given by (2.38) below.

$$f_{\rho_{\max},\text{wheel}}(\phi) = \text{abs}\left(\frac{r\,\varphi_{max}}{l\,\sin\phi + \cos\phi}\right) \tag{2.38}$$

Figure 2.12 shows an example of how this constraint limits the velocity space of a robot.

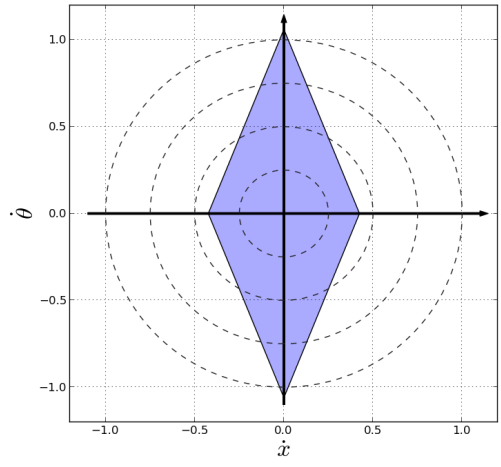


Figure 2.12: Example of how the drive wheel constant limits the available velocity space.

2.6 Higher Level Constraints

The previous sections formalized some of the constraints that limits the available velocity space for the two wheel configurations. In practice, the robot is a dynamic system, and the feasible velocities and motions will also be constrained by the dynamics of the robot and the temporal state. Analyzing the dynamics in details is usually very system specific. It depends on a several factors such as the motor characteristics, the current to the motors, the weight of the robot and external factors such as friction.

In this thesis I assume a very simple dynamic model for the robots, and simply constrain the maximum acceleration to fixed values. The translational acceleration is limited by a maximum acceleration $a_{drive,max}$ when increasing speed and and a maximum acceleration for breaking $a_{brake,max}$. The angular acceleration is modeled as being independent and limited by the value $a_{rot,max}$. The limits are defined below:

$$-a_{brake,max} \leq \ddot{x} \leq a_{drive,max} \quad \text{when } \dot{x} \geq 0 \tag{2.39}$$

$$-a_{drive,max} \leq \ddot{x} \leq a_{brake,max} \quad \text{when } \dot{x} < 0 \tag{2.40}$$

$$-a_{rot,max} \leq \ddot{\theta} \leq a_{rot,max} \tag{2.41}$$

2.7 Handling Constraints

The constraints described in the previous sections was formalized by a constraint function, where the maximum length of the velocity vector was found at a given curvature. Given a specific robot type, all these constraints can therefore be combined in a single constraint function. Below is a combined constraint function for a differential drive robot (2.42) and for an Ackermann robot (2.43).

$$f_{\rho_{\max},\text{differential}}(\phi) = \min \begin{cases} f_{\rho_{\max},\text{drive}}(\phi) \\ f_{\rho_{max},a_c}(\phi) \\ f_{\rho_{\max},\text{wheel}}(\phi) \end{cases} \tag{2.42}$$

$$f_{\rho_{\max},\text{Ackermann}}(\phi) = \min \begin{cases} f_{\rho_{\max},\text{drive}}(\phi) \\ f_{\rho_{max},a_c}(\phi) \\ f_{\rho_{\max},\text{steering}}(\phi) \end{cases} \tag{2.43}$$

2.8 Conclusion

This section presented the notation used for wheeled robots in this thesis. It presented a way to represent a velocity vector in polar coordinates. This representation enables a general way to describe robot constraints, as a constraint function giving the maximum length ρ of the velocity vector at a given curvature ϕ. The different constraints for both differential drive and Ackermann robots were described using this format.

3

Technologies

This chapter will provide an analysis of the different technologies that are currently used, or can potentially be used to build autonomous robotics solutions. Since autonomous robots often integrates many different technologies, the quality and robustness of a robot highly depends on the state of each of these. Each subject will be briefly presented with focus on how it actually works, price, and future potential in relation to robotics.

3.1 Wheeled Robot Platforms

3.1.1 Willow Garage PR2

The PR2 robot is a research and development platform created by Willow Garage Inc. a robotics research lab located in Menlo Park, California. It was developed through their Personal Robotics Program, and initially released in 2010 through the PR2 Beta Program, where Willow Garage lent out 11 Beta versions of the PR2 robots to selected research labs around the world. The program included a two-year commitment to pursue their research goal using the platform and open source software. Later in 2010, the PR2 went officially for sale with a price of $400,000 for regular buyers, and $280.000 through an open source 30% discount.

In 2011 they released a cheaper version with only one arm with a regular price of $285.000 and $200.000 through the open source discount.

Figure 3.1: The Willow Garage PR2 Robot

Source: www.willowgarage.com

The PR2 is a large robot, about the height of a human, and a 68cm square base. It includes two 7-DOF arms, a height adjustable spine, and a omni-directional wheeled base. Several perception and depth sensors are located in the tiltable head, in the gripper, and in the mobile base. It has powerful on-board processing-power provided by two separate computer systems each equipped with two Quad Core Xeon processors, 24GB ram, and 500GB HD. It utilizes an EtherCat bus for real-time control and also includes Gigabit Ethernet and WiFi. With the on-board 1.3kWH battery the robot is capable of running for approximately two hours. The mechanical hardware is higly modular, making it possible to swap grippers, change sensors or even the arms. It is controlled by the open-source Robot Operating System (ROS).

The aggressive roll out of PR2 robots through the PR2 Beta program quickly resulted in a large community of robotics researchers based on ROS and PR2 robots. Together with the aggressive development pace

Figure 3.2: The PR2 robot playing pool

at Willow Garage, the ROS system now includes an impressive amount of state-of-the-art functionality and libraries for perception, navigation and manipulation for the PR2. Many novel demonstration applications were built with the PR2, including one where the robot was playing pool (Figure 3.2), or a setup where it was making pancakes. With the massive specifications and price tag though, the PR2 is mainly useful as a research platform and not for creating practical consumer products.

3.1.2 Care-O-bot

The Care-O-bot is a mobile robot assistant developed at the Fraunhofer Institute for Manufacturing Engineering and Automation in Stuttgart, Germany. It is a result of more than ten years of development. The third generation version, the Care-O-bot 3, is built from industrial components, has a product like design and is capable of working in everyday environments. It has an omni-directional mobile base with four wheels, an on-board computer, three laser scanners and a 300 kWh battery.

The torso has a rear mounted arm and a tray that can be shoved away on the side. The torso can bend back and forth for simple user feedback or

Figure 3.3: The Care-O-bot 3 developed by Fraunhofer IPA

for better positioning the sensor head. The sensor head is either equipped with two cameras in a stereo setup, a time-of-flight camera, or a Microsoft Kinect. The standard arm is a 7-DOF light weight arm equipped with a three finger gripper with tactile sensors. Both produced by Schunk.

Figure 3.4: Schunk Dextrous Hand equipped with tactile sensors

These parts are off-the-shelf components from robotics hardware manufacturer, thus highly mature and high quality. Other grippers or arms can be mounted instead. A flexible casing can be attached to the body

structure resulting in a more comfortable appearance.

The parts in the current version of the actual robot are handmade, resulting in a very high cost in the same league as the PR2 robot. But by moving production to more industrial manufacturing, the Fraunhofer IPA hopes to lower the costs significantly opening up the potential to use the Care-O-bot platform for consumer applications.

3.1.3 KUKA youBot

The KUKA youBot is a mobile manipulator platform designed for research and education manufactured by the industrial robot manufacturer KUKA. It is made up by two main parts, the mobile platform and a robot arm.

The KUKA youBot mobile platform includes four motorized Swedish wheels, making it omni-directional. The platform hosts the power and also has an on-board PC, currently an Intel Atom Dual Core processor, 2GB Ram and 32GB SSD storage. The KUKA youBot arm is a 5 DOF arm with a two finger gripper attached. From base to tip, the arm with the gripper is 65.5 cm long. It is indented to be mounted on the mobile platform, and controlled by the on-board PC, but alternatively it can run stand-alone and controlled through an Ethernet cable.

Even if the two parts are designed to be used together, they can also be bought individually, or in a version with two arms. The current list price of the different configurations are listed in table 3.1, excluding a 10% discount eligible for universities and research.

Configuration	**List Price**
Mobile platform only	11.990,00 EUR
Arm only	15.990,00 EUR
Mobile platform with one arm	23.990,00 EUR
Mobile platform with two arms	38.990,00 EUR

Table 3.1: List price of different youBot configurations **Source:** youbot-store.com

Figure 3.5: The KUKA youBot with omni-directional mobile platform and one mounted arm

The robot is shipped with multiple open-source software that can be used to control it. This includes a high level driver for the robot implemented as a object oriented C++ API. In addition, they also include a control system based on ROS, with ROS configuration models for the youBot, a ROS wrapper for the youBot API, and a simulator setup.

3.1.4 Turtlebot

Turtlebot is a low cost mobile robot kit built using off-the-shelf robot components. It is an open source hardware project, meaning the design is publicly available and anyone are free to make, modify, or sell the hardware based on the design. The first version of the Turtlebot (Figure 3.6) was built on top of an iRobot Create mobile base, while the newest verion, the Turtilebot 2 (Figure 3.7), is built using a iClebo Kobuki from Yujin Robot.

The iClebo Kobuki is a mobile base designed for education and research. It includes a power supply for an external computer, provides

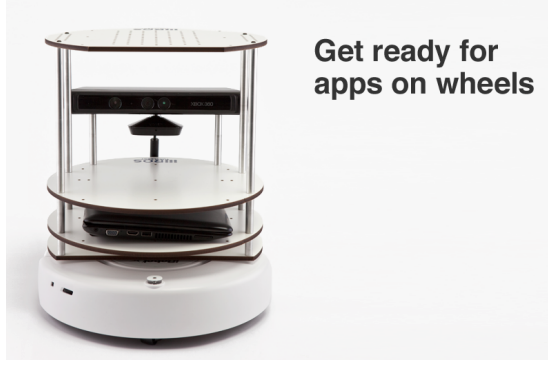


Figure 3.6: The First Generation TurtleBot using an iRobot Create

precise odometry, and has a variety of sensors such as gyroscope, cliff sensors, wheel drop sensor, and bumpers. The iClebo Kobuki can be used individually, but the TurtleBot 2 kit provides a the Microsoft Kinect and a mechanical frame for mounting the Kinect, mounting additional sensors and actuators, and a platform for carrying a laptop.

Figure 3.7: The Turtlebot 2 using an iClebo Kobuki

The Turtlebot is originally designed at Willow Garage and thus from the start a natively supported robot platform by ROS. Through ROS, the TurtleBots comes with a complete development environment. This including libraries for visualization, planning, perception, control and many demo applications

The Turtlebot 2 can be bought as a core kit with the base and the mounting frame priced as 749.00 EUR. The complete kit includes also a Laptop and the Kinect and is priced at 1299.00 EUR.

3.1.5 Pioneer P3-DX

The Pioneer P3-DX (Figure 3.8) is a popular mobile research robot that have been used in research for many years, but has been updated with a more powerful microcontroller and larger payload. It is a medium sized, two-wheel differential drive robot, designed for indoor laboratory or classroom use. It comes with 8 forward facing sonar sensors, wheel encoders and hot-swappable batteries. It has a built in microcontroller running the Adept ARCOS Firmware (Advanced Robotics Control and Operations), which makes the robot controllable from the packaged Pioneer SDK. Optionally, the robot can be equipped with an industry-grade computer with a Dual Core 2.26 GHz CPU, 8GB RAM and SSD hard drive. The robot can reach speeds of 1.6 $\frac{m}{s}$ and carry a payload of up to 23 kg.

Figure 3.8: The Pioneer P3-DX

The Pioneer P3-DX is sold by Adept MobileRobots [1], which has a wide range of research robots for both indoor, outdoor and underwater environments. The list price of the P3-DX robot is $4500, where univer-

[1] www.mobilerobots.com

sities are subject to a reduced price of $4000.

3.2 Sensor Technologies

For a robot to be able to perform any autonomous behavior it must be able to sense the environment. The information given by the sensor should be fast and reliable enough to use them to decide proper actions. Sensing should both be in relation to the actual task it is performing, e.g. detecting and picking up a specific object, or the environment all together to be able to navigate without collisions or generally stay out of trouble.

This section will focus on sensor technologies capable of sensing the environment i 3D, since it it an essential requirement for a mobile robot to navigate.

Other sensing problems such as object recognition, are usually task specific, and out of scope of this chapter.

3.2.1 3D Vision

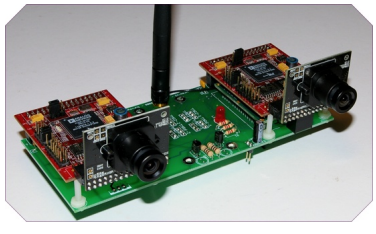
Humans and animals have the ability to perceive the world in three dimensions using eye sight. In computer science a similar ability can be implemented by performing advanced image analysis of images from digital cameras.

Passive 3D cameras make use of the fact that a 3D scene is seen differently when observed from different viewpoints. By extracting the same features from the two views of the scene, the depth can be extracted. 3D stereo can be coarsely classified in sparse and dense stereo. Sparse stereo algorithms extracts few higher level features, resulting in fewer depth measurements, but potentially providing higher level knowledge about what is seen in the scene. Dense stereo extracts and compares many low level features, giving a much more fine grained depth map.

Figure 3.9 shows two examples of stereo cameras in different price ranges. Figure 3.9b shows a Bumblebee 2 produced by PointGrey. It

is a fixed pair of cameras, pre-calibrated against mis-alignment and distortion, and interfaced through Firewire. It is sold in several different configurations with varying focal length, and resolution with a list price of either 1560 EUR (640x480) or 1990 EUR (1024x768).

A cheaper alternative is the Stereo Vision System (SVS) from Surveyor shown in figure 3.9a. It consists of two Surveyor SRV-1 Blackfin Cameras supporting up to 1280x1024 resolution, and also includes processors for data-processing, motor controllers and Wifi connection. The Surveyour SVS firmware is open source, and it is intended to be used by researchers, educators and developers for enabling 3D vision to robotics. The Survey SVS system costs $550.



(a) Surveyor SVS **Source:** www.surveyor.com

(b) PointGrey Bumblebee®2 **Source:** www.ptgrey.com

Figure 3.9: Stereo cameras

3.2.2 Time-of-flight Cameras

A time-of-flight-camera (ToF) uses the known speed of light to measure the depth in an image, by measuring the time an infrared light pulse emitted from the camera takes to hit the object and return to the camera sensor. Some types of cameras uses fast laser diodes or diodes behind electronic shutters to emit very short pulses, other uses modulated light and measures the phase change. Due to the fast speed of light, the measurements can be obtained very fast, resulting in high frame rates. Since the light is collected with a 2D image sensor, the distance is measured simultaneously for each point in the image.

Figure 3.10: Swiss Ranger SR4000 ToF Camera from Mesa Imaging

Traditionally ToF cameras have been very expensive. The Swiss Ranger SR4000 (Figure 3.10) produced by Mesa Imaging[2] costs approximately 4000$.

Recently some cheaper alternatives have become available, driven by the need for gesture recognition in games and consumer multimedia devices. SoftKinetic[3] is a company focused on gesture recognition. It started in 2007 by a team of mathematicians, 3D imaging specialists, software engineers and game enthusiasts. They produce two depth sense cameras based on ToF technology. The DS325 (Figure 3.11a) is sold for $249 and designed for a range measurement of $15cm$ to $1m$, with a resolution of 320x240, and running at up to 60fps. It is interfaced and powered through a USB connection and has a power usage of maximum 2W. The related D311 (Figure 3.11b) camera is sold for $299 and the depth measurements has a resolution of 160x120. This camera also has a "far" mode where it measures in a range of $1.5m$ to $4m$.

[2] mesa-imaging.ch
[3] www.softkinetic.com

(a) DS325

(b) DS311

Figure 3.11: ToF Cameras from SoftKinetic. **Source:** www.softkinetic.com

3.2.3 Structured Light

Light Coding is a 3D sensing technology provided by PrimeSense, an Israeli company funded in 2005. It is mostly famous for being patented to Microsoft to use in their motion sensing controller Kinect[Mis13] for the Xbox 360 video game console. When the Kinect was released in November 2010, it delivered unseen 3D sensing performance in the consumer price range. Today the technology has been implemented in many other products such as:

- Asus Xtion, a motion sensor controller targeted at PC application and games[Asu13]
- Matterport 3D Scanner, a portable scanner to build colored 3D models of interior spaces[Mat13]
- iRobot AVA, a mobile tele-presence robot for use in medical environments[iRo13]

A 3D sensor utilizing Light Coding technology consists of an infrared light source, a CMOS image sensor capable of reading the infrared light, and optionally a second CMOS image sensor to provide gray-scale or color information about the scene (Figure 3.12). The light source projects a pseudo-random light pattern into the scene invisible to the human eye. Since the infrared CMOS sensor observes the scene from a slightly different angle, the light pattern will appear distorted as a result of the

depth in the scene. Thus depth information can be derived from the distorted light pattern.

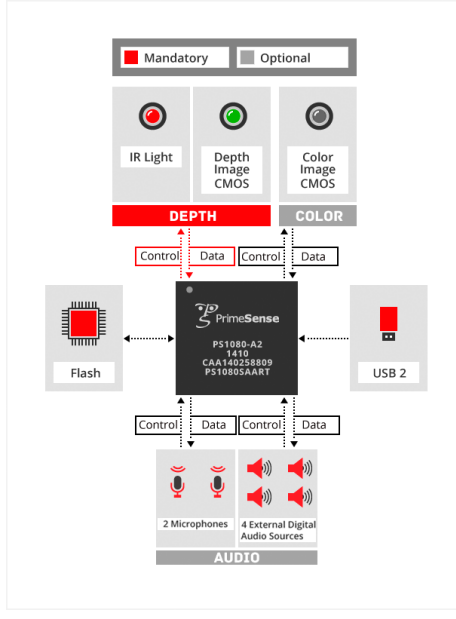


Figure 3.12: PrimeSense Technical Overview **Source:** www.primesense.com

PrimeSense has patented the Light Coding technology, including the pseudo-random pattern they use, and methods to recognize the movement of human bodyparts from the depth information. PrimeSense also supplies several SoC (System on a Chip) solutions implementing depth sensing, human body part recognizing, and multimedia interfaces. The SoC solution is clearly one of the strengths in the technology since it makes it easy and low cost to use the technology in new products, and enables PrimeSense to keep implementation details secret

3.3 Conclusion

This chapter described a list of examples within mobile robot platforms and sensor technology. Most of the robot platform were designed for research, very complex and therefore very expensive. Concerning the price

of sensor technology, it is clear to see how powerful a factor the gaming and consumer multimedia marked is. The Kinect based on PrimeSense, and the SoftKinetic depth cameras, are examples of cheap and powerful sensor technology driven by the consumer market.

4

Generic Navigation

4.1 Introduction

The most fundamental task for wheeled robots is navigation. In the simplest form the task consist of moving the robot from one position to another. In practice, navigation includes quite complex situations, such as deciding a path, following it safely, detecting and handling unexpected obstacles or following a moving target. In addition, the navigation algorithm must take the geometric and dynamic constraints of the robot into account.

Due to this large complexity, the navigation task is usually divided into a set of subproblems such as:

- Localizing
- Trajectory Planning
- Trajectory Following
- Obstacle Detection and Avoidance
- Learning Maps

Mobile robot navigation has been the subject of research for decades. An early example is a team at the LAAS laboratory in Toulouse, France

who already in 1977 investigated the design and control of mobile robots and built the Hilare robot. It was a three wheeled robot equipped with camera and a laser-scanner mounted on a tiltable platform, in addition to several ultrasonic sensors. The robot platform was the base for early work in autonomous navigation and obstacle avoidance such as [Cha82][Mor80].

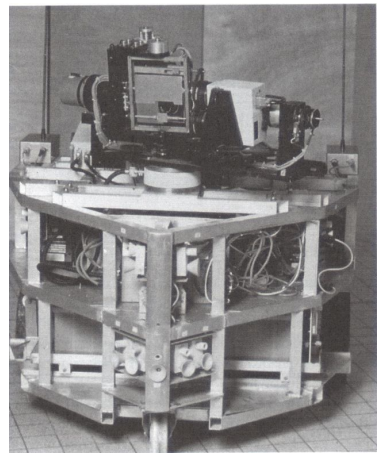


Figure 4.1: Hilare robot was build at LAAS laboratory in Toulouse, France. The project started in 1977.

The author in [Cro85] presents a navigation system for an intelligent mobile platform, called IMP built in 1984. It is based on a pre-learned model of a finite global environment, as well as a dynamically maintained model of the local environment integrating measurements from a rotating range sensor and touch sensors. The estimated position is corrected based on the pre-learned model and the current measurements. The system is capable of global planning and local obstacle avoidance. Other examples of work from the same period includes [Kha86] and [Sch87], both covering planning and execution in dynamic environment.

Today, almost 30 years later, many consider the planning, mapping and navigation problems for indoor robots somehow solved, and have moved onto more challenging research areas such as robot swarms and

aerial robots. Advancements in robotics, sensor technology and computer hardware, means that much more advanced algorithms based on probability theory is used for navigation today. Implementations for localization and mapping algorithms such as FastSlam [Hae+03] or GMapping [GSB05][GSB06] are available[1].

The Robot Operating System (ROS) includes a complete and somehow general navigation package for omni-directional and differential drive robots. It takes odometry information and sensor measurements, and outputs velocity commands to send to a mobile base. The navigation package in ROS has been used to control a variety of robots with miscellaneous results, since it assumes many properties of the robot. The robot should preferably to be close to round or squared, and capable of rotating on the spot. The robot must be running ROS, the odometry model is constrained, and it must have a laser scanner or other range sensors mounted on the front of the robot base. The command interface is limited to a target pose the robot should move to.

This example of a navigation algorithms with a lot of assumptions about the robot is not atypical. The motivation to create something truly generic seems to be missing. This most likely explains why novel and advanced research in higher level problems in robotics navigation often is based on easy to control wheel configurations such as omni-directional or differential drive with the wheels in the middle. Such configurations are easy to control because the orientation has little influence on planning and collision checking. This choice makes many navigation implementations simpler, but unfortunately also difficult to implement on more practical shaped robots such as differential drives robots with wheels mounted in the back, or rectangular Ackermann robots.

4.1.1 Contribution

This chapter presents the work towards a purely generic navigation solution for wheeled mobile robots motivated by the following goals.

[1] openslam.org

- Generic: Works for different types of robots
- Configurable: Parameters maps to geometric properties of the robot
- Predictable: Well defined where the robot will drive
- Safe: Avoid fatal collisions

Based on a survey of existing methods and algorithms the chapter presents the following:

- A generic way to represent a trajectory
- A method to convert the trajectory so it can be driven in a smooth motion
- A method to create a safe velocity profile for the robot
- A path following controller

4.2 Survey

4.2.1 Localization

Localizing is the problem of determining the the pose of the robot based on available data such as sensor reading or control output. Most localization methods are based on a mixture of 1) odometry calculations based on input from wheel encoders and the known geometry of the robot and 2) periodic corrections based on sensors. Using odometry means that the the position can be updated with low computational cost and high frequency, but odometry results in cumulative errors that grow significantly in short time. Further details about calculating odometry can be found in [DJ00].

To be able to correct the cumulative odometry errors, robust localization methods must implement additional sensors. A GPS sensor can provide absolute positioning although with with a low update frequency

but with absolute errors of several meters. Sensor input from a laser-scanner or camera vision can be matched to features in a known map to derive the robot pose. Choosing the best localization method for a given robot application depends highly on the available sensors, the required precision, available computational time, and the environment where the robot operates (indoor or outdoor). It is therefore difficult to design a general software-only solution, thus the actual localization problem will have low focus in this thesis. The further navigation study will assume the following properties on the localizer:

- High frequency localization updates are available based on odometry, providing a single estimated pose and an estimated error.
- Low frequency corrections will occur, potentially resulting in significant changes to the estimated pose.

4.2.2 Trajectory Following

Litterature has proposed several control strategies for making a robot follow a specified trajectory.

One category is based on a look-ahead strategy where a point is chosen somewhere in front of the robot on the trajectory, and the robot is controlled to drive towards it. One of the earliest examples of a robust implementation of the look-ahead strategy is the pure pursuit algorithm in [Cou92]. It calculates the curve that will guide the robot back on track, and was implemented on several outdoor vehicles. Several parameters defines how the point is chosen, so the algorithm both can guide the robot on the trajectory, and handle situations where the robot is far away from the trajectory.

A slightly more recent example is the Dynamic Window Approach [FBT97]. It limits the search space of velocities based on the dynamic properties of the robot, and choses a most optimal solution based on a mixture of objectives such as driving towards the goal and avoiding obstacles. The Dynamic Window Approach has shown to work well in

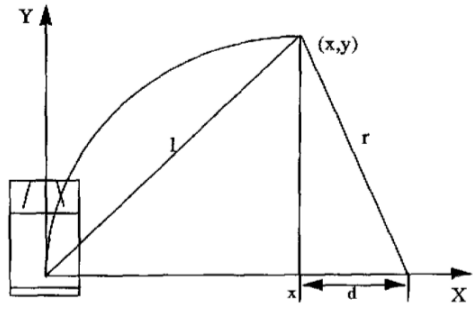


Figure 4.2: Pure pursuit calculates the curved motion that will drive the robot towards a give target point (x, y) on the trajectory. **Source:** [Cou92]

practice and is used for the navigation implementation in ROS. It is also based on a look-ahead strategy, where an immediate local goal is chosen somewhere on the trajectory and used to guide to robot. Possible velocities are analyzed and the best one is selected based on a weighted objective function taking heading towards the goal, avoiding obstacles and maximizing speed into account. Figure 4.3 shows the search space in the velocity space.

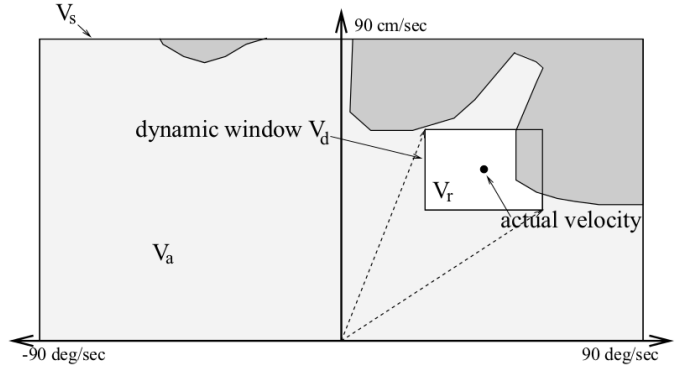


Figure 4.3: The Dynamic Window Approach analyzed a limited window around the current robot velocity in the velocity space. Selecting a optimal velocity based on the target point and detected obstacles. **Source:** [FBT97]

The look-ahead strategy is very intuitive for humans, since it is similar

to how we drive a car, and results in what appear to be very smooth motions. In practice, the strategy is not well suited for the requirements listed in section 4.1 for the generic navigation because: (1) It allows the robot to deviate from the planned trajectory when there are obstacles nearby, thus the behavior is often unpredictable. (2) It has problems handling sharp turns, rotation on the spot and also reverse motion, since it relies on a target a certain distance in front of the current position.

Another widely used strategy is to control the robot to follow the trajectory with a closed loop feedback controller. Following the trajectory precisely and safely requires at minimum: (1) A control law that minimizes the error and keeps the controller stable. (2) A trajectory that is feasible to follow with respect to the robots kinematic and dynamic constraints. (3) A strategy to update the reference point on the trajectory.

Such a controller will follow the specified trajectory even if it turns out to be blocked by obstacles. Therefore the collision avoidance must be handled at a higher level, and the trajectory updated on the fly to guide the robot around obstacles. Assuming a proper localization method, it is always predicable where the robot will drive, and where it will be guided in the near future.

Trajectory following controllers are addressed in a lot of literature. A linear control law that drives a robot towards a certain reference pose, including the orientation, is found in [SNS11]. Based on the position error ρ, the angle towards the target position α and the orientation error β the control laws in (4.1) and (4.2) drives the robot towards the goal position.

$$\dot{x} = k_\rho \, \rho \tag{4.1}$$

$$\dot{\theta} = k_\alpha \, \alpha + k_\beta \, \beta \tag{4.2}$$

A different approach is found in [Mic+93] where the authors derive a control law for unicycle-type and two-steering-wheels robots. It controls the angular velocity $\dot{\theta}$ to make the robot follow the path. The error input is the current sideways distance to the path, and assumes that the

forward velocity $\dot{x}$ is decided elsewhere. This controller is extended by [LL03] taking modeling uncertainties into account, and using a virtual target that moves with a predefined velocity along the path as controller reference point. In [GK08] the control law is used to guide a wheelchair, where smooth and comfortable motions is important.

[VAP08] presents a set of control laws originally designed for guiding marine vehicles in formation, but in [CAG08] it is generalized for wheeled robots. One controller moves the virtual target along the trajectory to make sure the vehicle can keep up, and the second guides the vehicle towards the virtual target. A noticeable difference compared to the earlier control law is that input to the second controller includes the current velocity of the virtual frame. The control law corrects only the positional error.

4.2.3 Inevitable Collision States

An inevitable collision state (ICS) describes an irrecoverable state for a robot system, in which it is impossible to avoid a future collision. No matter what the control input or future trajectory followed by the system is, a collision with an obstacle will eventually occur. The term was first mentioned in [LKJ01] and was later formally defined and investigated in [AFF04].

Consider a wheeled robot driving with high speed towards a solid wall. If the speed of the robot is too high for the robot to be able to stop before reaching the wall, and it is also unable to steer away to avoid colliding with the wall, the robot is in an ICS. Clearly, the ICS space is dependent on several properties:

1. The current state of the robot including pose and velocity
2. The kinematic and dynamic properties of the robot
3. Obstacles in the environment

In [AFF04] it is claimed that studying the ICS space will help avoiding collisions and result in safer robot systems. The strategy is of course

to avoid ever entering a ICS. Initially, taking the ICS space into account when planning a trajectory for a wheeled robot will make sure the trajectory is safe for the robot, with respect to all the obstacles known beforehand. When driving the trajectory though, unexpected obstacles might suddenly appear in front of the robot, e.g. when passing a corner, making it enter a ICS and eventually collide. The solution mentioned in [AFF04] is to assume the worst case scenario, and consider only the field of view of the robot as obstacle free when planning. Thus the robot will either drive slowly when passing corners closely, or drive far away from the corner to optimize the field of view. Collisions with unexpected obstacle appearing at execution time can thus be avoided. The method is denoted 'safe motion planning'.

Figure 4.4 illustrates a simplified view of the different categories of state space. If the robot enter an actual ICS state it will eventually end up in one of the contained collision states. The method to assume the worst-case-scenario will grow the unavailable state-space even further as the figure illustrates.

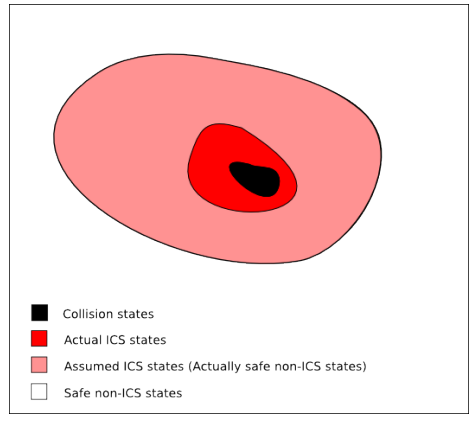


Figure 4.4: ICS

4.3 Curve Theory

4.3.1 Bézier Curves

A Bézier curve is a two dimensional polynomial curve, parameterized by a series of two dimensional control points. The number of control points minus one defines the order of the curve, thus an n-order curve is parameterized by the points $[\mathbf{P}_0, .., \mathbf{P}_n]$.

Definition

Let $\mathbf{B_{P_0P_1...P_n}}$ denote the Bézier curve parameterized by $[\mathbf{P}_0, .., \mathbf{P}_n]$.

Then a Bézier curve of order n can be defined recursively by two Béizer curves of one lesser order:

$$\mathbf{B_{P_0}}(t) = \mathbf{P_0} \tag{4.3}$$

$$\mathbf{B_{P_0P_1...P_n}}(t) = (1-t)\mathbf{B_{P_0P_1...P_{n-1}}}(t) + t\mathbf{B_{P_1P_1...P_n}}(t) \tag{4.4}$$

$$t \in [0, 1] \tag{4.5}$$

where t defines a point on the curve where $t = 0$ is the start and $t = 1$ the end.

A Bézier curve has several notizable properties:

Well defined start and end

The points $\mathbf{P_0}$ and $\mathbf{P_n}$ defines the start and the end point of the curve respectively, thus this type of curve can be used to interpolate between two points.

Easy derivatives

The derivative of a Bézier curve of order n, is a Bézier curve of order $n-1$ parameterized by the points:

$$\mathbf{P'_0} = n(\mathbf{P_1} - \mathbf{P_0}) \quad (4.6)$$
$$\mathbf{P'_1} = n(\mathbf{P_2} - \mathbf{P_1}) \quad (4.7)$$
$$\dots \quad (4.8)$$
$$\mathbf{P'_{n-1}} = n(\mathbf{P_n} - \mathbf{P_{n-1}}) \quad (4.9)$$

Convex Hull Property

The Béizer curve will always be contained by the convex hull defined by the control points.

Convertible to higher order

A Béizer curve of order n is equal to the order $n+1$ curve with control points $[\mathbf{P'}_0, .., \mathbf{P'}_{n+1}]$ where:

$$\mathbf{P'_k} = \frac{k}{n+1}\mathbf{P_{k-1}} + (1 - \frac{k}{n+1})\mathbf{P_k} \quad (4.10)$$

4.3.2 Calculating Curvature

The signed value of the curvature of a point t on a Bézier curve is found by:

$$\kappa(t) = \frac{\det(\,\mathbf{P'}(t), \mathbf{P''}(t)\,)}{||\mathbf{P'}(t)||} \quad (4.11)$$

4.4 Generic Trajectory Representation

The trajectory defines the path the robot must follow through space to move from one pose to another. Often in previous work, such as in [LSB09], the trajectory function denoted Q maps into only the position of the robot $Q(t) = (x, y)^T$ which is adequate only if the orientation has little influence on collision e.g. if the robot is round or square shaped. In a generic navigation implementation this property can not be assumed

so the trajectory function must map into the full 2D pose of the robot: $Q(t) = (x, y, \theta)^T$.

4.4.1 Parameterization

When planning the initial collision-free path with respect to the static obstacles in a map, the trajectory is independent of the time variable t. Instead it is parameterized with a time independent variable. The most popular methods is to use the Cartesian distance s traveled in the forward direction by the robot from a starting point on the trajectory. This method is used in [LSB09], [SG91] and [HK07].

A pose on the trajectory is thus given by a function $Q(s)$ where:

$$s = f_s(t_1) = \int_{t=t_0}^{t_1} \dot{x}_r(t)\,\mathrm{d}t \tag{4.12}$$

For the generic approach in this thesis the parameterization with respect to Cartesian distance traveled showed to be inadequate. Trajectories for some types of robots includes rotation on the spot, where no Cartesian travel is happening. In addition, the navigation should robustly handle trajectory paths that include reverse robot motion where $\dot{x} < 0$.

To solve this limitation the work in this thesis uses a more generic parameterization. The solution is to parameterize the trajectory with respect to the integrated length P of the velocity vector ρ (see (2.10)). The resulting trajectory function is of the form $Q(P)$ where:

$$P = f_P(t_1) = \int_{t=t_0}^{t_1} \rho(t)\,\mathrm{d}t \tag{4.13}$$

The value of ρ is always positive, and only zero when neither Cartesian or angular motion is taking place. Thus any value of P where $P_{start} \leq P \leq P_{goal}$ uniquely identifies one and only one position on the trajectory.

4.4.2 Segments

In practice the complete trajectory can be a complex shape, some places very curved and other consisting of straight line segments. Therefore it is not feasible to represent the complete trajectory function mathematically especially not for handling general cases.

A popular solution is to define the trajectory piecewise using individual segments connected together in places called knots. The parameters of each of the segments can be adjusted independently, providing a high degree of freedom in specifying the shape of the trajectory.

To create a closed trajectory the knots must at minimum connect the segments in the same position and orientation. For a robot to be able to follow the trajectory in a continuous motion, the line curvature must also be continuous in the segments and in the connecting knots. Non-continuous jumps in the curvature requires unlimited acceleration to follow, thus in practice the robot will have a jerky motion, and result in instability.

The simplest form for trajectory segments consists of straight line segments between a series of intermediate poses. Many path planners for mobile robots such as the one found in [Lik+05] returns a path based on straight line segments. Straight lines can only interpolate the position between the poses, not the orientation. Rotation only happens in the actual knot points, thus following such a trajectory requires stopping in each knot point.

In [FBT97] the trajectory is modeled with circular arcs, where each segment has a constant curvature. Such a trajectory appears smooth, but all the knots that connect segments of different curvature will have non-continuous curvature.

The authors in [MF07] and [Sah+07] use cubic Bézier splines to model the trajectory. These can be adjusted to provide first order continuity (velocity) in the knots. As seen from (4.11) curvature also depends on the second order derivative (acceleration), thus curvature continuity cannot be guaranteed using cubic Bézier splines.

The authors in [GK08], [SG91] uses fifth order B-splines to specify the trajectory. B-splines is a generalization of a Bézier curve and they are widely used to model smooth curves and surfaces in computer graphics, where first order continuity is sufficient.

The additional requirement for continuous curvature can be fulfilled using B-splines of proper order, but the method used by the authors means that local changes in the trajectory affects up to six neighboring segments.

The authors in [LSB09] uses fifth order Bézier splines to model the trajectory segment. They present a method and a series of heuristics that makes it possible to match both the first and second order derivative in the knots, by only adjusting the two neighboring segments. The beginning trajectory is a sparse series of positions without orientation that are found to be collision-free when connected by a straight line. The method iteratively optimizes the trajectory into a curved trajectory, while not straying too much from the straight line shape.

The ability to modify the trajectory and still maintain a continuous curvature by only local adjustments is an important property for the generic navigation implementation. Many situations such as newly discovered obstacles, large pose corrections by the localizer, or manual overrides, can result in the need to constantly modify the trajectory. The more local the changes can be kept, the fewer new poses have to be collision checked, thus updates can be faster and occur more frequently.

This thesis adopts part of the method from [LSB09], with some modifications to increase generality:

- Extended to be able to model rotation-on-the-spot segments.
- Input is a collision-free, possibly curved trajectory with pose information continuous in position and orientation.
- Knots are curvature-matched without the need for the second derivative to be the same value.

- Allow for knots where the robot is in full stop to be non-continuous in curvature.

4.4.3 Curvature Matching

As mentioned in section 4.4.2 the curvature should generally be continuous along the trajectory for the robot to be able to follow it in a smooth motion. There are two special situations where continuity is not required: (1) The knot connecting a segment with forward motion, and a segment with reverse motion. The robot is at a complete halt in the knot so contentious curvature is not required. (2) A rotation-on-the-spot (ROTS) segment will have infinite curvature, thus a connecting segment will be non-continuous. The robot is required to come to a complete halt in the knot. Table 4.5 lists the different types of segments and whether they must be connected by continuous curvature or by putting the robot in a complete halt.

	Forward	**Reverse**	**ROTS**
Forward	Continous	Halt	Halt
Reverse		Continous	Halt
ROTS			Continous (Always ∞)

Figure 4.5: Connecting Knots

This thesis proposes a method to convert a trajectory consisting of 3rd order Bézier curves with continuous position and orientation, into a trajectory of 5th order Bézier curves with continuous curvature.

Initial Trajectory

It is assumed that the initial trajectory has been found and is:

1. Collision Free.

2. Continuous in the position and orientation.

3. Modeled as 3rd order Bézier curves. (Except ROTS segments)

The scope is not to find the actual trajectory, it is assumed to be planned beforehand. Approximating any curved trajectory by 3rd order Bézier curves is trivial, since it is one of the most widely uses for third order Bézier splines in computer graphics.

The initial trajectory, denoted $Q^{(3)}$ (4.14), consists of a series of segments $\hat{Q}_i^{(3)}$. Each segments is either a curve segment, modeled by the four points describing a 3rd order Bézier curve (4.15a), or a ROTS segment, modeled as a point, and a start and end orientation (4.15b). The superscript "(3)" denotes that it is using third order Bézier curves.

$$Q^{(3)} = < \hat{Q}_0^{(3)}; \hat{Q}_1^{(3)}; \dots ; \hat{Q}_n^{(3)} > \tag{4.14}$$

$$\hat{Q}_i^{(3)} = \begin{cases} < P_{i,0}^{(3)}; P_{i,1}^{(3)}; P_{i,2}^{(3)}; P_{i,3}^{(3)} > & \text{Curve (a)} \\ < P_i; \theta_{i,0}; \theta_{i,1} > & \text{ROTS (b)} \end{cases} \tag{4.15}$$

Curvature Matching Method

The method from [LSB09] matches two segments by setting both the first and second order derivative to the same value. Because the initial trajectory is continuous in the orientation, the direction of the first derivative is continuous in the knots, while the magnitude may be different. The velocity at which the trajectory is driven is independent of this magnitude, so this discontinuity is not a problem. In this method, the value of the first derivative is conserved, and only the second derivative is modified to curvature-match the two segments. This better maintains the shape of the original curve.

Denote the curvature in the start-point of segment $\hat{Q}_i$ as κ_i^{begin}, and similar in the end-point of segment $\hat{Q}_{i-1}$ as κ_{i-1}^{end}. The values of these curvatures are found from (4.11). An average value for the curvature is found by a weighed average, inverse proportional to the length l of each of the segments respectively. The weighted average is used because the second derivative of longer segments can be adjusted more without too much differences in the original shape.

$$\kappa_{avg} = \frac{l_{i-1}\,\kappa_i^{begin} + l_i\,\kappa_{i-1}^{end}}{2(l_{i-1} + l_i)} \tag{4.16}$$

The value of the second derivative of the two initial segments are denoted $\mathbf{a}_{i-1}^{end}$ and $\mathbf{a}_i^{begin}$. Since the second derivative is a two dimensional vector, first a unit vector for the average direction is found by:

$$\mathbf{a}_{unit} = \frac{\mathbf{a}_{i-1}^{end} + \mathbf{a}_i^{begin}}{||\mathbf{a}_{i-1}^{end} + \mathbf{a}_i^{begin}||} \tag{4.17}$$

The length of the second derivative that will result in a curvature of κ_{avg} is found by inversing (4.11) and isolating the length. This results in the following expression:

$$a_{len} = \frac{||\mathbf{v}||^2\,\kappa_{avg}}{\frac{\mathbf{v}}{||\mathbf{v}||} \cdot \mathbf{a}_{unit}} \tag{4.18}$$

where $\mathbf{v}$ denotes the first derivative in the respective end-point.

Continuous-Curve Trajectory

The continuous curvature trajectory $Q^{(5)}$ is similar to the initial counterpart, except the curved segments is modeled by six points describing a 5th order Bézier curve. The ROTS segments are not changed.

$$Q^{(5)} = < \hat{Q}_0^{(5)}; \hat{Q}_1^{(5)}; \ldots; \hat{Q}_n^{(5)} > \tag{4.19}$$

$$\hat{Q}_i^{(5)} = \begin{cases} < P_{i,0}^{(5)}; P_{i,1}^{(5)}; P_{i,2}^{(5)}; P_{i,3}^{(5)}; P_{i,4}^{(5)}; P_{i,5}^{(5)} > & \text{Curve (a)} \\ < P_i; \theta_{i,0}; \theta_{i,1} > & \text{ROTS (b)} \end{cases} \tag{4.20}$$

The two control points in each end of the Bézier curve is found by using (4.10) to imitate the initial curve as closely as possible.

$$P_0^{(5)} = P_0^{(3)} \tag{4.21}$$

$$P_1^{(5)} = \frac{2}{5}P_0^{(3)} + \frac{3}{5}P_1^{(3)} \tag{4.22}$$

$$P_4^{(5)} = \frac{3}{5}P_2^{(3)} + \frac{2}{5}P_3^{(3)} \quad (4.23)$$

$$P_5^{(5)} = P_3^3 \quad (4.24)$$

$$(4.25)$$

With this procedure the value of the first derivative in the end points is not changed.

The two middle control point can be placed to control the value of the second derivative denoted $\mathbf{a}_0$ and $\mathbf{a}_1$ in the two end points. The expressions for the two middle control points are derived for the equations for Bézier derivatives listed in section 4.3.1.

$$P_2^{(5)} = \frac{1}{20}\mathbf{a}_0 + 2P_1^{(5)} + P_0^{(5)} \quad (4.26)$$

$$P_3^{(5)} = \frac{1}{20}\mathbf{a}_1 + 2P_4^{(5)} + P_5^{(5)} \quad (4.27)$$

Using the above expression all the segments in the initial trajectory is converted into a curvature-continuous set of fifth order Bézier segments.

4.4.4 Example Trajectory 1

Figure 4.6 shows a trajectory for a differential drive robot. The trajectory is defined by six different poses, that put together results in the five segments listed in table 4.1. The robot starts in the lower left corner, drives forward for $60cm$, rotates 90° clockwise on the spot and drives forward $30cm$. When it has passed the wall, it turns right while driving forward, and comes to a halt in the lower right corner. From here it reverses $60cm$ until it is parked with the back to the wall.

The trajectory is represented by third order Bézier segments and one ROTS segments, thus includes all the possible types of segments and knots.

The curvature of the initial trajectory is illustrated in figure 4.7 as a blue curve. The knot connecting segment 3 and segment 4 is the only one where the robot does not have to come to a complete halt, but from

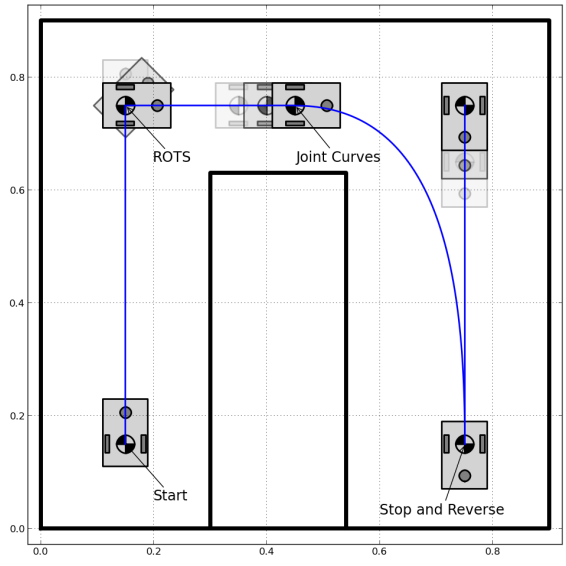


Figure 4.6: Example Trajectory 1

the figure one can see than the curvature is non-continuous in this knot. Using the above method, the trajectory is modified to apply continuous curvature in this knot by modifying the two connecting segments. The curvature of the modified trajectory is illustrated by the green line in the figure, where one can see how the two connecting segments have been modified, and the curvature is now continuous.

#	Start Pose	End Pose	Segment Type
1	$(0.15, \quad 0.15, \quad \frac{\pi}{2})$	$(0.15, \quad 0.75, \quad \frac{\pi}{2})$	Forward
2	$(0.15, \quad 0.75, \quad \frac{\pi}{2})$	$(0.15, \quad 0.75, \quad 0.0)$	ROTS
3	$(0.15, \quad 0.75, \quad 0.0)$	$(0.45, \quad 0.75, \quad 0.0)$	Forward
4	$(0.45, \quad 0.75, \quad 0.0)$	$(0.75, \quad 0.15, \quad -\frac{\pi}{2})$	Forward Curve
5	$(0.75, \quad 0.15, \quad -\frac{\pi}{2})$	$(0.75, \quad 0.75, \quad -\frac{\pi}{2})$	Reverse

Table 4.1: Poses and Segments of Example Trajectory 1

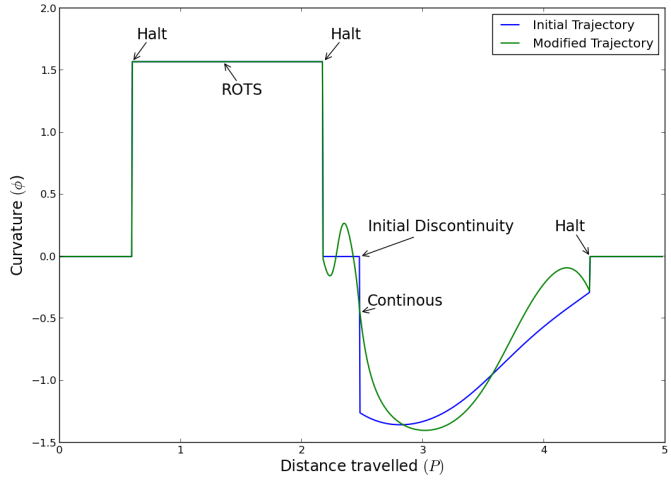


Figure 4.7: Curvature of Example Trajectory 1

4.4.5 Example Trajectory 2

Figure 4.8 shows a trajectory for an Ackermann robot driving slalom around a series of round obstacles. Due to the steering angle constraint the last turn is too sharp to perform in a simple motion. Instead the robot stops, reverses shortly, and continues around the obstacle.

The blue line in the figure illustrates the initial trajectory defined by 3rd order Bézier curves. The curvature of the initial trajectory is illustrated by the blue curve in figure 4.9. From the figure one can see that the trajectory has several places with non-continuous curvature.

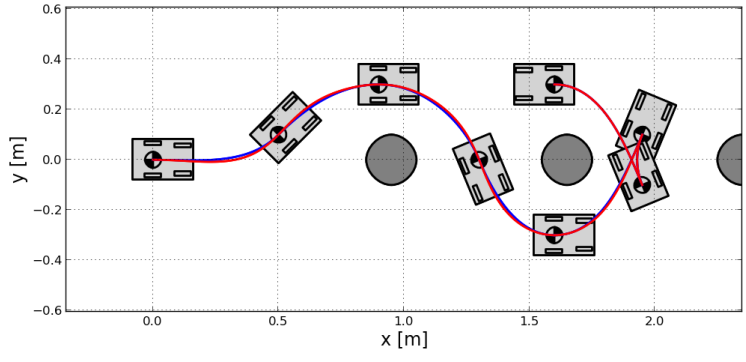


Figure 4.8: Example Trajectory 2

Therefore the above method is used to convert the trajectory into a continuous curvature version with 5th order Bézier curves. The red line in figure 4.8 shows the modified trajectory, and how it has been slightly changed.

The green curve in figure 4.9 illustrates the curvature for the modified trajectory. It can be see that is continuous, except in the two places where the robot changes between forward and reverse motion and is at a complete halt.

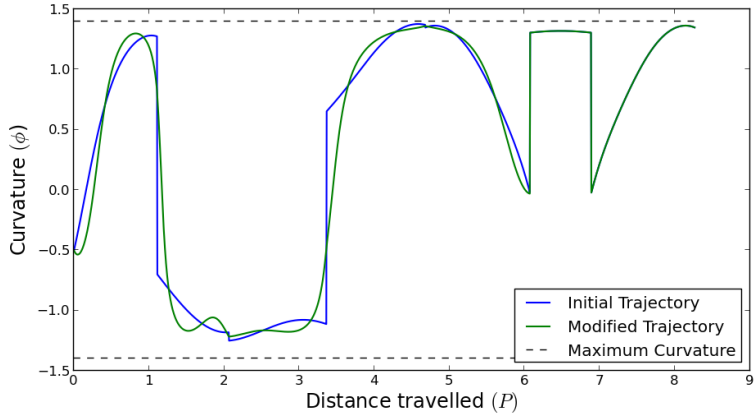


Figure 4.9: Curvature of Example Trajectory 2

4.5 Generic Trajectory Following

4.5.1 Velocity Profile

The trajectory from section 4.4 is purely geometrical. It has been modified to be able to drive with only the required stops, but it contains no information about the actual speed the robot should have.

When following the trajectory, the robot must obey the velocity and acceleration constraints, and also brake in proper time before full stops and curves. To be able to handle this, the trajectory is analyzed in a three step process, resulting in a velocity profile (4.28) of the maximum allowed velocity ρ_{max} the robot can have at any location P on the trajectory. The maximum velocity is defined as a maximum length ρ of the velocity vector (2.10), thus limiting both the translational and the rotational velocity.

$$\rho_{max} = f_{vel}(P) \tag{4.28}$$

For simplicity, each trajectory segment is split into 20 sub-segments in which constant acceleration is assumed. A maximum velocity is calculated for each boundary point between two sub-segment. The three step process is:

1. Find the maximum velocity based on only the curvature at each boundary point, using the constraint function (2.42) or (2.43).

2. Iterate forward through the trajectory. Find the maximum obtainable velocity from the previous boundary point and using maximum acceleration. Limit it to the value from step 1.

3. Iterate backwards through the trajectory. Find the maximum allowed velocity limited by the future sub-segment boundary and with maximum deceleration. Limit it to the value from step 1.

4.5.2 Example Trajectory 1

The velocity profile for trajectory 1 in the previous section has been calculated using the above method. The velocity profile after step 1 is illustrated by the red curve in figure 4.10. It is seen how it follows the magnitude of the curvature. The reason why the value of p is high in the areas with high curvature is, that is also has a $\dot{\theta}$ component that makes it larger even if the forward velocity $\dot{x}$ is lower. The green curve shows the profile after step 3, where the velocity is limited before and after the robot has to drive slowly or come to a complete halt.

4.5.3 Example Trajectory 2

The velocity profile for trajectory 2 is illustrated in figure 4.11 and shows a similar behavior. The maximum velocity after step 1 follows the magnitude of the curvature, since it is purely derived from this curvature. The velocity profile after step 3 now has a few places where the robot is configured to drive slower, including the beginning acceleration, the end stop, and the three-point turn.

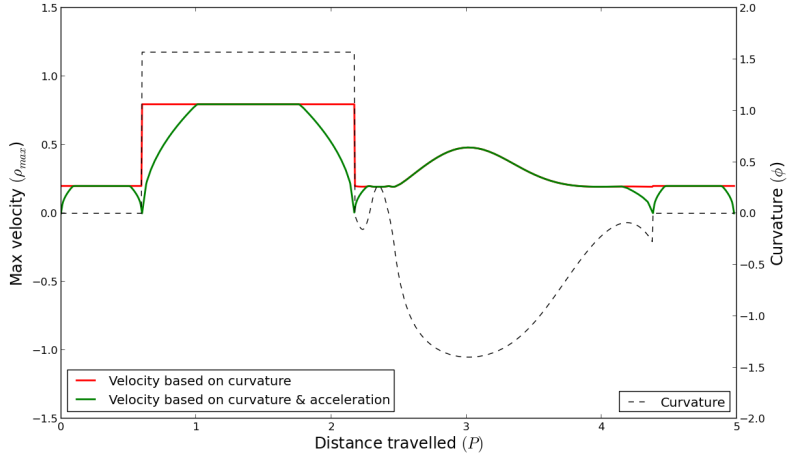


Figure 4.10: Velocity Profile for Example Trajectory 1

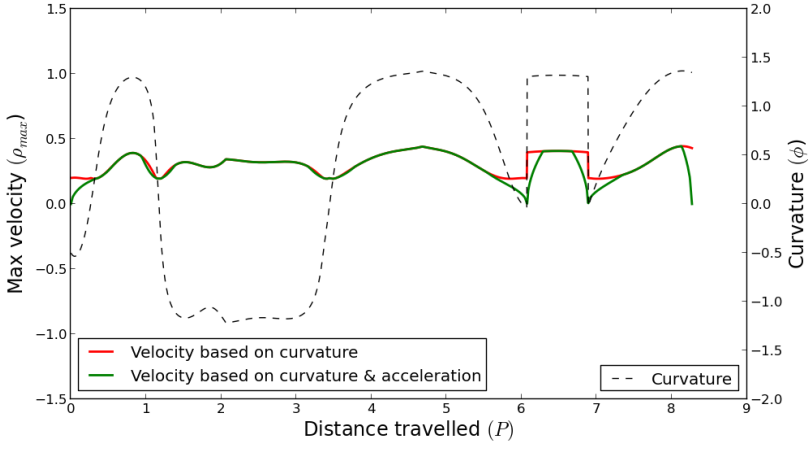


Figure 4.11: Velocity Profile for Example Trajectory 2

4.5.4 Controller

Unfortunately, none of the trajectory following controllers mentioned in section 4.2.2 handles the situation where the robot is reversing. In addition, even if they are designed to handle curved trajectories, most only use the position error, and thus can not handle rotation on the spot situations. To support a generic navigation implementation, and to perform the experimental tests in this thesis, a control law with both of these properties was needed.

Fortunately most of the control laws adheres to the same interface. They take the current robot pose and a moving virtual reference target, and outputs a desired robot velocity.

Therefore the solution was to design a simple control law that supported reversing and rotation on the spot, and use it to show that a robot can follow a trajectory based on the calculated velocity profile. It should have the following properties:

1. It should correct both the position and orientation errors.
2. Be simple and tunable. It is only supposed to work as a starting point and to show that following the trajectory based on the velocity profile is possible.
3. Adhere to the common interface, where input is the robot pose and a virtual target frame, and output is a robot velocity.

A control law was designed based on a mixture between the equations from (4.1) and (4.2) and the more complex control law in [CAG08].

The input to the controller is the desired pose reference ξ_{ref} given in the same reference frame as the pose of the robot. The reference velocity $\dot{\xi}_{ref}$ is the local velocity of the reference point on the trajectory.

$$\xi_{ref} = \begin{pmatrix} x_{ref} \\ y_{ref} \\ \theta_{ref} \end{pmatrix} \tag{4.29}$$

$$\dot{\xi}_{ref} = \begin{pmatrix} \dot{x}_{ref} \\ 0 \\ \dot{\theta}_{ref} \end{pmatrix} \tag{4.30}$$

$$\tag{4.31}$$

The error in the reference is transformed into the frame of the robot using its currently known orientation.

$$e = \mathbf{R}(\theta)(\xi_{ref} - \xi) \tag{4.32}$$

$$= \begin{pmatrix} e_x \\ e_y \\ e_\theta \end{pmatrix} \tag{4.33}$$

The controller is designed to handle small errors, that means when the robot is on track on the trajectory. It is not designed to guide the robot back on the trajectory if it has lost track.

The controller equations (4.34) and (4.35) consists of four elements.

- The reference velocity $\dot{\xi}_{ref}$ is mapped directly into the robot velocity.
- The correction of the positional e_x error controller by a proportional gain κ_ρ.
- The correction of the sideways error e_y controlled by the gain κ_α. Since this error is corrected by turning the robot in the direction towards the trajectory, the correction is also proportional to the expected positional velocity $\dot{x}_{ref}$. This makes the robot turn towards the trajectory even when reversing, and also avoids over correction when the robot is driving slowly.
- The orientation error e_θ is corrected with a proportional gain κ_β.

$$\dot{x}_{control} = \dot{x}_{ref} + \kappa_\rho \, e_x \tag{4.34}$$

$$\dot{\theta}_{control} = \dot{\theta}_{ref} + \kappa_\alpha \, e_y \, \dot{x}_{ref} + \kappa_\beta \, e_\theta \tag{4.35}$$

In case of cumulative position errors, the robot can appear to be on track based on the current pose estimation but in reality be far from the trajectory. When the localizer corrects the pose, the error will become large, resulting in a desperate corrective velocity from the robot. To have more control of the motion that guides the robot back on track, the trajectory is modified when large localizer corrections occur with the current robot pose as the starting point. This is the same method used in [LSB09].

4.6 Experimental Results

An experiment was performed to verify that it is possible to follow the continuous curvature trajectory with the analyzed velocity profile. The robot used for the experiment is a small differential drive robot called SMR (Small Mobile Robot) designed and used locally at the Department of Electrical Engineering. The robot has two powered rear wheels (diam $= 65mm$), and two free caster wheels in front. The dimensions of the robot is $28cm \times 32cm$ (w $\times$ l). The robot is seen in figure 4.12.

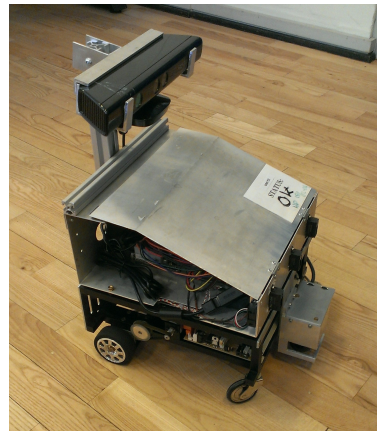


Figure 4.12: The DTU SMR Robot used for the experiments

The controller in section 4.5.4 was used to drive to robot to follow the example 1 trajectory. The control loop was run at 100Hz on a local

computer connected to the robot through a LAN cable. The gains used for the controller were $\kappa_\rho = 2.5$, $\kappa_\alpha = 10.0$ and $\kappa_\beta = 4.0$. Figure 4.13 shows the experiment results. It illustrates both the reference pose and the actual path of the robot.

It is seen from the figure that the robot nicely follows the path, although with slight oscillations. When following the curve, after approximately 15 seconds it is seen that the robot is slightly off track both in position and in orientation. These observations could be a result of the simple controller, non-optimal parameters for the controller, or high delays over the network. Despite these problems, the results show how the robot nicely follows all parts of the trajectory including rotation on the spot, full stops, reversing, and curved motion. The robot slows down before the full stops and it drives slower during the curved motion as the velocity profile dictates.

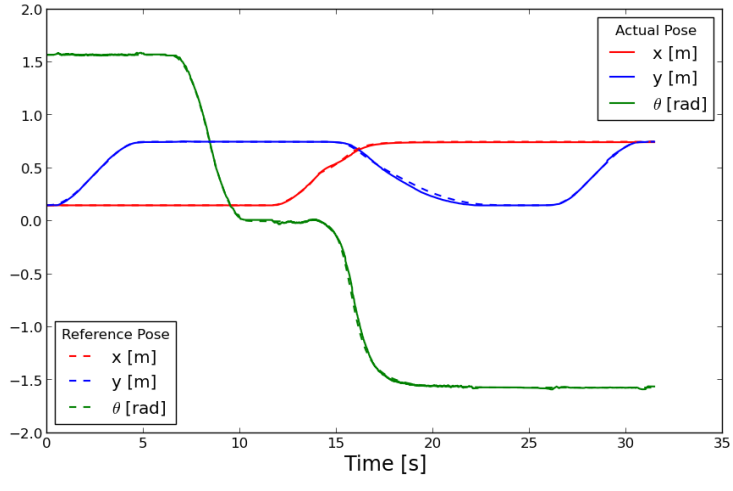


Figure 4.13: Experimental results for the SMR following Example Trajectory 1

4.7 Conclusion

In this chapter the initial work towards a generic navigation solution for wheeled mobile robots was performed. The survey clearly showed that such a generic solution was needed. For example, despite being branded as generic, the navigation package in ROS is designed primarily for omni-directional robots with round or square shape.

The survey showed that research with navigation genericity as the primary motivation was very limited. Many different method and implementations for navigation exists, but often they are designed for a specific type of robot and motivated by higher level research goals. As a result, the work towards a generic solution started on a very low level, with trajectory representation and trajectory following.

Based on previous work in [LSB09], it was decided to represent curved trajectories using fifth order Bézier curves. The work in this thesis enhances the trajectory representation, so it is possible to represent rotation-on-the-spot and reverse motion. A method to convert an existing curved trajectory into a drivable version with continuous curvature was presented and verified with two example trajectories.

The chapter also presented a method to create a velocity profile for such a trajectory. The velocity profile defines the maximum velocity for both translational and angular velocity, the robot is allowed to have at any point in the trajectory. This maximum velocity adheres to the dynamic constraints of the robot, and also takes the future part of the trajectory into account, so the robot will slow down in proper time. The velocity profile was derived for the two example trajectories.

In the survey, no trajectory following controller was found that was capable of guiding a robot through a trajectory with rotation-on-the-spot and reverse motion. Therefore a custom control law was created to exist as an initial example, and to perform experiments with a real differential drive robot driving through one of the example trajectories. During the experiment the robot followed the trajectory, braking before full stops and driving slower through the curve. At the curved part of

the trajectory the robot was slightly off track, showing that the controller could potentially be improved.

The results shows that the methods are usable for implementing a generic navigation solution for both differential drive and Ackermann robots.

4.7.1 Future Work

The method to create a drivable trajectory assumes that a curved trajectory adhering to the geometrical constraints of the robot is already found. A navigation solution must also include the step of finding this trajectory.

The method to make the trajectory curvature-continuous slightly alters the trajectory. Even if the initial trajectory is feasible, the method could therefore potentially generate a trajectory with collision, or increase the curvature beyond the geometrical limits of the robot. One potential solution is to allow the method to also move the knot points slightly, to optimize the path and minimize the curvature in high-curvature areas of the trajectory. When the trajectory is modified, the method should perform collision checking. Moving the knots will also allow for other optimization schemes, such as with respect to driving speed or higher safety.

The controller for trajectory-following is designed as very simple. Further work is intended to be put into analyzing the performance of the controller, or potentially consider a more advanced version. It is also possible that less generic controllers should be designed for the different types of wheel combinations instead of a generic one. Doing so, will make it possible to take the respective robot model further into account, and potentially implement observers for detecting errors in the configured robot parameters.

5

Robotics Middleware

5.1 Introduction

When designing control software for robot systems one often encounters the concept of a control framework. Sometimes it is denoted a middleware, since it is software with the purpose of connecting other software implementation and libraries together. The different types of middleware scales from simple support network libraries, to full scale ecosystems with code generators, integrated test frameworks, visualizers and build tools.

The overall purpose of the framework is to support a user in the development of his system. By taking care of low level responsibilities such as data exchange and threading, he can focus on the functionality implementation where his expertise is high. Many frameworks share the optimistic vision of making functionality implementation sharable and reusable between several users, departments or even on a global scale.

An optimal middleware should be able to support the developer in his most optimal work-flow. It should provide enough support to free the user from low-level tedious and error prone tasks, but still provide enough freedom so he is not unnecessarily constrained when implementing his solution.

5.2 Survey

During the years a substantial number of control frameworks and middleware implementations have sprung out from the need of a better fundamental for building robots. Most projects have been an effort from a single university and research group, and have never really expanded beyond this border. The scope of this survey is not to cover all of them, but to focus on a few of the more successful ones that have gained a large group of users and are still in use today.

Several more extensive surveys of robot middlewares exists, such as [KS07].

5.2.1 Orocos

The Orocos[Bru01] project (Open RObot COntrol Software) dates all the way back to 2001 where it was started as a relatively small EU sponsored project with the aim of creating an open source control framework for use in the field of robotics research. The motivation was that the commercial robot frameworks often used in research at that time didn't expose the lower level sensor data with high enough efficiency to be used in advanced research.

Since then, Orocos has been through many release cycles, and has matured into a respected and widely used framework in robot control. It now consists of several sub-projects including RTT (The Realtime Toolkit), KDL (Kinematics and Dynamics Library), and BFL (Bayesian Filtering Library).

The Realtime Toolkit is the actual middleware, created in C++, and supports the implementation of a real-time control system separated in individual components. The component model allows several communication interfaces such as synchronous data flow, asynchronous events, commands, and RPC calls (Figure 5.2). The data-exchange is setup by creating connections between components, creating a network of components with well defined dependencies and data-flow. This model fits very well to implement control loops, but it also requires the user to deal

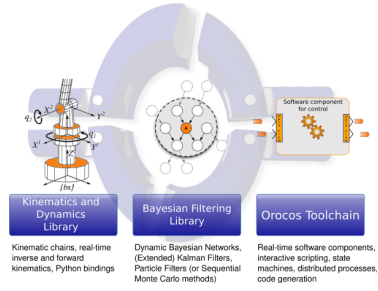


Figure 5.1: Orocos Logo **Source:** www.orocos.org

with the low level task of defining the connections. Practical experience shows that control engineers find this level of control very suitable, while non-control engineers prefer the connections to be handled in a more abstract manner. Being a middleware for implementing control loops, RTT supports hard real-time guarantees of timing when running components on an operating system that supports it.

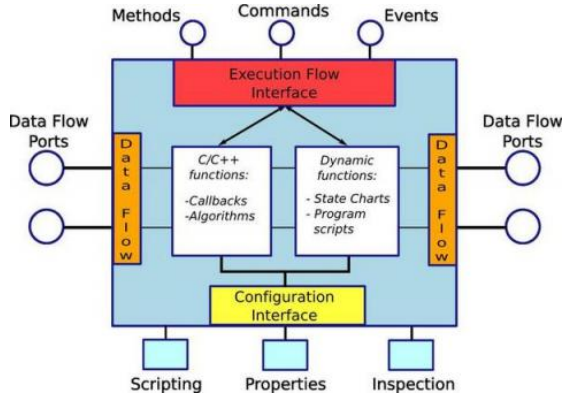


Figure 5.2: Orocos Component Model **Source:** www.orocos.org

5.2.2 Player/Stage

As the name implies, the Player/Stage [Ger+01][CM05] project consists of two sub-projects. Player provides a server architecture, where a set

of generic interfaces for e.g. sensors, actuators and higher level functionality can be implemented by devices. Devices are independent on each other, and by registering on the server, they functionality become available to remote clients that connect to the server through a socket interface. Since the server devices and the clients are separated by a socket interface, it has been possible to make the client library available in several programming languages such as C, C++, Python, and Ruby.

Stage is a graphical simulator for multiple mobile robots, and implements simulated versions of Player devices. Since the simulated devices in Stage implements the same interface as the real hardware devices, they can be easily exchanged.

The fact that Player is natively network based, and provides an easy to use simulator, has made it very powerful for controlling multiple mobile robots. They have been capable of targeting both advanced research and educational applications, since clients are available in several programming languages.

5.2.3 ROS

ROS was originally denoted a Robot Operating System, and includes much more than just a middleware. It was designed to meet the challenges encountered when designing large-scale service robots in the STAIR project [QBN07] and the Willow Garage Personal Robots Program [Wyr+08]. Some of the design goals were to be peer-to-peer, distributable on multiple hosts, and support for multiple programming languages.

A ROS system consist of a series of asynchronous nodes which runs as a separate process, and exchange data using network connections. The data must be of the form of ROS messages which is a hierarchical composition of a series of primitive data-types. They are defined in a ROS specific IDL (Interface Descriptor Language), and there-after converted into code implementations in the available languages. As such, data exchange is platform and language independent, and ROS systems can thus scale to large and complex distributed systems.

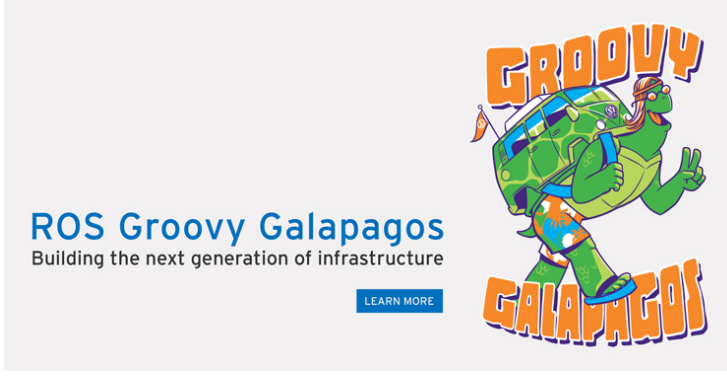


Figure 5.3: ROS Groovy Galapagos Banner **Source:** www.willowgarage.com

ROS includes its own build environment, code generators, and a huge variety of tools for visualization, debugging, logging and assisting the user. The user-friendly tools, together with an ever growing library of control functionality implemented in ROS, and a large effort into supporting the users through an open source community, ROS has gained an impressive user group and a very active community.

5.3 Development Process

Software is often designed by software engineers for other software engineers. As a result, many libraries and middlewares are designed with main focus on fixing the problems that the designed himself has experienced, or that he thinks other people are struggling with. In practice the middleware will be used by a huge diversity of different people, with different backgrounds, different expertise and focusing on different areas of the value chain.

The choice between a simple to use middleware for unexperienced developers, and a powerful and complex middleware for experts is not strict. It is a challenge to find the best compromise between these two choices, and there will always be arguments in both directions.

5.3.1 Stakeholders

Being high technology with practical use, the use of robotics spans widely from research to industry. Researchers are motivated by creating publishable results, or proof-of-concept systems to argue that their novel methods are better than previous work. Industry are motivated by practical results, systems that are working and usable, and that can either give them an advantage in contrast to competitors, or expand the market potential. In practice, one often encounters that there is a big gap between the products from research and the products expected by industry.

To close this gap, a robotics middleware should be usable by both research and industry. Making it easier, faster and require less resources to use the results from a research project in a practical commercial application.

The study has identified three overall categories of users. Technology Researchers, Research Integrators and Industrial Users. In pracise many people will have different responsibilities during a project, and this might fit into several of these categories at different times, or in different research areas.

Technology Researchers

A robot application consist of a wide diversity of technologies, such as complex mechanical design, hardware integration, advanced low level control and a wide range of intelligent high level autonomous behaviour. Many of these technologies are unrelated, thus much research only focuses on one or few areas of interest where expertise is built and novel research published.

For this user-group, having to use a robotics middleware can be a unnecessary burden. Novel research and simulated tests, or tests based on recorded experimental data is faster to perform in mathematical environments such as MATLAB[1] or SciPy[2]. This frees the researcher from

[1] http://www.mathworks.com
[2] http://www.scipy.org

practical problems such as programming complexity, timing issues and deployment.

Researchers are motivated by being able to publish his results, which requires description of his method and test results, but not necessarily a reference implementation. Practical tests are often only performed on a single or a small number of reference robot platforms that are available to the research group, and for which they have experience. In the field of his expertise the researcher will have the best knowledge of how his algorithms or methods are adjusted and configured for a specific robot.

The technology researcher could optimally provide or assist in the following:

- Detailed description of his algorithm or method, e.g. in a research paper.
- A reference implementation.
- Information about whether his methods can be used for other robots and how to modify and configure them for specific cases.

Research Integrators

Obviously, only running algorithms and tests in MATLAB or similar environments is of limited interest compared to controlling a real world robot. Getting a robot running requires integration of many different research technologies, and relying and using work provided by other researchers. A user who needs to perform this integration of other peoples work is denoted a technology integrator.

Integration in research can be a time-consuming and ungrateful job. Time is spent on understanding other peoples work, and fighting tedious and practical problems with few publishable results. Maintaining only a few reference platforms in a research group can minimize the time spent on integration, but neglecting a platform for only short periods can make it very difficult to make it usable again.

Experience has shown that unreasonable amount of time is spent on integrating technologies into new robot applications. Even simple problems with well known solutions such as simple control, odometry calibration, sensor integration and simple localization are time consuming to set up on a new robot. One often fights the same problem over and over again, due to small differences, resulting in unnecessary cost of resources but also an enormous source of errors.

A robot middleware can help with this, and the magic keyword that is thrown around is "reuse".

Discussions and arguments in favor of "reusability" often focuses on implementation reuse, which can be reuse of actual source code, compiled libraries or installable packages. Fortunately the methods for code-reuse has been very mature in other open source communities for many years, and the procedure is quite straightforward. It requires a certain amount of effort towards proper implementation, testing, documentation and subsequent maintenance and support.

While code reuse can save the integrator a lot of precious time, he is still left with a secondary task that can be equally time consuming and error-prone. The algorithms and libraries must be configured for the specific robot, task and environment, and integrated properly into the rest of the system.

Traditional software libraries optimally provides a simplified interface to the user, hiding unnecessary implementation specific details, thus the user can focus on his specific use-case and not on how the underlying algorithms work. Many hours of work spent of integrating reference implementations of control algorithms and different methods and technologies provided by other research groups, into our own robot applications shows that this is rarely the case for robotics software. The underlying algorithms are highly advanced and based on many difficult to understand properties such as probability. Important parameters relate directly to core properties of the algorithms, and unless one know the algorithm detail, it is impossible to choose the correct parameter values.

A practical example is the ROS Navigation stack[3], a software package providing mapping and autonomous navigation for wheeled robots. It is branded as a easy-to-use package and provides a well documented guide explaining how to integrate it with your robot system running ROS. In many aspects this package is a inspiring example in reuse, since it tries to provide a solution to a very generic and common task in wheeled robots. Experience has shown that in practice people find it much more difficult to use than expected. Especially if they are using it for robots with different properties than the Willow Garage PR2[4] robot it was originally designed for. The following reasons have been identified:

- The guide describes how to modify **your** system to fit the very strict interface of the navigation node, not the other way around.
- The package contains a huge amount of implementations for planning, control and mapping. It is difficult to get a overview of the best choice for your application, and the quality of each.
- The parameters are very implementation specific. Such as map dimension in pixels, or the number of particles for the Monte Carlo localizer. Setup is covered by a "tuning" guide.
- Many things are hard-coded. Such a one specific odometry error model.
- The external interface is very strict and simple. You simply provide a goal.
- It internally includes too high level behavior, such as recovery behaviour and planning strategy. Thus you get little control over this at runtime.

[3] http://www.ros.org/wiki/navigation
[4] http://www.willowgarage.com/pages/pr2/overview

Industrial Integrators

Industrial integrators are users who require integration of the technology for industrial or commercial applications. In addition to the same challenges as the Research Integration, they are usually more constrained in relation to time, resources and the hardware the software should run on. For this group, is is also important to know the quality and robustness of the available implementation, since the requirements for robustness and reliability is higher.

5.3.2 Use Cases and Functionality

The scope of this section is not to create an excaustive list of use cases and requirements for a robotics middleware. Instead it will elaborate of a few of the more exotic cases not supported very well by the existing frameworks. Either because they are very specific for the robotics research process, because inadequate solutions exist in the existing frameworks or because they are new requirements from the industrial stakeholders.

Rapid Research Prototyping

As mentioned in section 5.3.1 technology researchers often prefer a more practical and mathematical working environment such as MATLAB or SciPy in Python. As such, the initial reference implementations occur in these languages and not necessary in the native language of the middleware. Being able to interface these prototype implementations directly with the "real" control system of the robot has several advantages. It avoids the need for tedious and error-prone re-implementations, and makes it possible to perform experiments early in the research process.

Distribute on Multiple Hosts

Robot systems are very modular. Having to run the complete control system on one single computer or microcontroller is a model that rarely fit in practice. They often consists of different actuators e.g. robot arms,

wheelbases or speakers, and a huge variety of sensors. This modular approach also spans into the controller hardware, where different computers control sub-parts of the systems. Prototype algorithms are easiest run on a desktop PC, while the rest of the control system run on the robot. Many ready-to-use implementations of network communication exist that can be used to connect the different parts manually. But this easily results in user-defined and undocumented protocols. Obviously, the network communication and system distribution is a low level functionality that most users should not have to worry about. Instead it should be handled natively by the middleware.

Unconstrained Data-flow Abstraction

The core responsibility of middleware is to connect, synchronize and exchange data between the different components that make up the system. The available patterns for connecting and synchronizing has a big impact on how free the user is to split his functionality into components and obtain a powerful architecture. The use case does not define any particular pattern, only to stress that the available data-flow patterns should not put unnecessary constraints on how the user can define and split up his system.

Unconstrained Data-types

When designing a control system that is only used in a single software group of research department, it is easy to dictate which libraries and data-types the users should use. Some middlewares have tried to perform the same dictation and limit the data-types that can be used. An example is ROS that provide a set of data-types, so called ROS messages, that must be used for data exchange. They do provide an IDL (Interface description language) to create user defined data-types, but they can only be built by a combination of their data-types. As a result, a running ROS system often includes an enormous amount of data-type conversion.

Internally, components and libraries often use more complex data-

types that are optimized for the given purpose and the implementation language. Even if two components in ROS use the same internal representation of an image, it must be converted into the ROS message version of an image to be transported, and then back again when delivered. Obviously, this constraint works against the user, instead of supporting him in his work. Often several different implementations of data-types exists to represent the same type of data. A middleware should be unconstrained in what data-types it is capable of exchanging and provide transparent conversion of different data-types that represents the same type of data.

Composition and Interfaces

While middleware can assist in splitting the control system into a multitude of smaller reusable components, in most cases, the components are only really usable in groups. Forcing the user to understand and configure other peoples components individually to deploy a higher level functionality is not optimal. With composition, many components can be grouped together to externally appear as one single component. Internal communication can be hidden and only the interface required to interact has to be exposed, making them much easier to understand and use. By defining a certain interface for a higher level functionality, several different component compositions can be used to implement the respective functionality. As long as they match the interface they can be interchanged.

Deployment and Lifetime Management

A complex control system is a dynamic entity with an entanglement of internal runtime dependencies. One part of the system needs a functionality or resource provided by a completely different part of the system to be available, before it can perform its own function.

Users often only consider the steady state, where all components are loaded and running. But it is not a straightforward process to go from a non-running system, to the steady state in an asynchronous system

where dependencies and dependees can start up randomly.

Deployment and lifetime management should handle the dependencies to provide a predictable startup and shutdown with respect to dependencies. In case of crashing or components otherwise becoming available, dependent parts of the system should also perform a controlled shutdown, until the respective components are either restarted or again available.

Low-Cost and Low-Power Hardware Support

In research, it is less important what type of computational power that is required to run the proposed methods and obtain usable results. The argument is often that computers has, and always will, continue to grow in power, and results that are difficult to obtain today will be generally available in few years.

Industrial stakeholders on the other hand needs the robot control system to run on todays hardware. They are much more constrained in cost, power consumption and physical space. Keeping the processing overhead and memory consumption low of both the middleware and its dependencies are important to insure that systems built with the middleware or small parts thereof, will be capable of running on low-end hardware.

5.4 Conclusion

The work in this project is focused towards being able to build consumer products from the available robotics technology. This results in several additional requirements on the middleware, compared to research only work. Some of these requirements results from practical constraints such as using embedded and low-power hardware. Others relate to increased need for reliability and predictability in the running control system, since industrial and consumer products will run for months and years, and failures and downtime can be expensive. At the same time, it is not desirable to use a middleware that is industry-only or proprietary, since it hinders the cooperation with academia.

The survey of existing middlewares showed several potential choices. The Orocos RTT appears to be very mature, uses a powerful component model that allows for code reuse and model-driven engineering. It also supports predictable hard real-time control and has several available models for connecting control systems over network. The synchronous and connection driven nature of RTT components follow the normal way of thinking in control theory, and it also appears that RTT is build for control engineers.

A second viable option is ROS. It is not only a middleware but is a complete eco-system with build system, release management, tools for debugging and visualizing and a huge variety of existing robot control implementations. The asynchronous network oriented model in ROS with distributed nodes and the publisher & subscribe patterns is a little further from regular control systems. But it appears many researchers finds this model easier to use and understand, and the huge community of ROS users is a very persuasive argument towards using it.

Based on these arguments, the choice fell on a mixture of Orocos RTT and ROS. The Orocos RTT is used for implementing low level control loops, and potentially run in realtime, while ROS is used for higher level functionality. This gives access to the whole ROS community, the powerful tools and the existing functionality.

5.4.1 Observed Issues

Unfortunately it was observed that several of the requirements mentioned in section 5.3.2 are not supported optimally by the ROS middleware. ROS is designed for research and high-end hardware running Ubuntu Linux, and putting it on more exotic low power embedded hardware turned out to be a challenge. Both due to performance issues, and because the very ROS-specific build system makes it difficult to cross compile for other architectures. Data exchange in ROS is constrained to a limited set of data-types, ROS-messages, or compositions thereof. Using the standard node model means all data will be pushed through the

network stack, even when transfered between two nodes on the same machine. This adds an enormous bottle-neck. Since ROS nodes exists as individual executables, the bring-up process is difficult to control since deployment happens by executing all these programs at once.

Due to all these limitations, mainly concerning lack of performance and predictability, it was concluded that the ROS middleware should not be used for serial industrial or commercial systems, unless they were only demonstrations or prototypes. During this thesis a substantial amount of work was therefore directed towards analyzing how the ROS middleware could be improved or replaced. Chapter 6 presents the design of the DARC middleware which is designed to be able to replace the ROS middleware in the ROS environment and fulfill the industrial requirements.

6

"DARC" Middleware

The previous chapter included an introduction to middleware and examples of existing middleware used in robotics. While ROS appeared to be the best choice, mainly due to high popularity in research, a large and active community, and a large selection of available functionality implemented in ROS, it had some disadvantages when used for commercial or industrial products. The large community is something that took years to build, and would not have happened if ROS was not a powerful system to use for researchers. One of the reasons that ROS is easy to use for researchers is that it is much more than a middleware. It includes a build and release environment and tools for navigating it. It includes tools for logging, visualization, and testing, and an environment for documentation and searching for assistance.

Within the scope of this project, two main components in the ROS system showed to be inadequate. The first one was the custom build-environment that meant it was very difficult to compile ROS systems for embedded systems. The second was the performance of the middleware.

Instead of just trying to modify the current systems, and create hackish workarounds, this project dug into the core problems and contributed to provide next generation replacements of these two system components. This resulted in a new build environment called "Catkin", and a prototype on a superior implementation of a ROS compatible middleware

called “DARC”.

6.1 The Catkin Build System

“Catkin” is a build system first used in the ROS release named Fuerte from April 23, 2012. It is based on CMake and Python to provide advanced dependency handling, and code generation in a native CMake style. It will be briefly described here because it was created in cooperation of Willow Garage as part of my external PhD stay in 2011 at Willow Garage, the company behind ROS. The effort this project put into “Catkin” was to solve a series of fundamental practical problems rather than research oriented. Despite this, it provides a very important fundament that, among other things, made it possible to create “DARC”.

catkin
catkin is the official build system of ROS and the successor to the original ROS build system, rosbuild. catkin combines CMake macros and Python scripts to provide some functionality on top of CMake’s normal work-flow. catkin was designed to be more conventional than rosbuild, allowing for better distribution of packages, better cross-compiling support, and better portability. catkin’s work-flow is very similar to CMake’s but adds support for automatic ’find package’ infrastructure and building multiple, dependent projects at the same time. The name catkin comes from the tail-shaped flower cluster found on willow trees – a reference to Willow Garage where catkin was created.
Source: *http://www.ros.org/wiki/catkin/conceptual_overview*

6.1.1 Motivation

ROS is made up by a large amount of different software packages. Some packages provide the core functionality, some include the tools, and the majority is the actual control systems and functionality implemented in ROS. The package model is essential to provide a high level of reusability,

and allows users with different needs to pull in only the parts of the system that they need. These packages dependends on each other, and in a practical ROS systems there is an enormous amount of internal dependencies to handle. Some are compile dependencies including C++ headers and libraries, some are tool dependencies, some are input to code generators and some are run-time dependencies between Python scripts.

The majority of users do not want to worry about handling these low level dependencies, therefore the initial ROS included a custom build system called "rosbuild". Each package is configured with: (1) A CMake build file containing mostly rosbuild specific CMake-macros, specifying how to compile the source files in the respective package. (2) An *xml* file listing the packages it depends on, and build information exported to dependent packages.

The initial "rosbuild" is very user-friendly for beginner programmers, which also means it is designed mainly to standard build requirements. The large amount of custom macros, and custom handling of dependencies, means that non-standard build requirements are difficult to handle. Cross compiling is the process of compiling a library or executable with the intention of running it on a different system (the target) than the one it is being compiled on (the host). Therefore it must also link against target-specific versions of all binary dependencies instead of using the ones installed on the host system. Cross compiling is the preferred method to compile for small embedded systems, since it is unnecessary and a waste of resources to put a complete compiler tool-chain on such a small system. The internal dependency handling of rosbuild was not designed to handle the separation of host and target dependencies, thus the main requirement of running ROS on embedded systems was unnecessary complex due to rosbuild.

Cross Compiler
A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. Cross compiler tools are used to generate executables for embedded system or multiple platforms. It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an operating system. It has become more common to use this tool for paravirtualization where a system may have one or more platforms in use.
Source: http://en.wikipedia.org/wiki/Cross_compiler

With the introduction of catkin, the dependencies were handled internally by CMake in a much more optimal way. Advanced ROS specific functionality such as sorting for dependencies, and code generation, were done by CMake hooking into a series of platform independent Python scripts. This resulted in:

1. Compilation times that were magnitudes faster compared to rosbuild.

2. Better cross-platform support that allowed for building on e.g. Microsoft Windows.

3. Made all CMake options available, including configuring for cross compilation.

4. More powerful separation of package dependencies.

The last item, meant that it was possible to use smaller parts of the ROS core library individually, e.g. ROS messages without the actual ROS middleware. As a direct result, it was now possible to create a new implementation of the ROS middleware, that would be compatible with ROS messages. This is exactly what “DARC” is.

6.2 Design Considerations

Attempting to create a complete replacement to the ROS middleware is no easy task. Yet it gives a unique possibility to make fundamental changes, and take many more requirements into account than just the performance and predictability issues. When designing ROS, many design choices were taken with respect to the currently available libraries and the requirements known at that given time. The current situation is not much different, except there is a large amount of practical experience and lessons learned from years of using ROS. There is a much more broad variety of users, and use-case that can be taken into account.

But expecting to be able to design the perfect middleware that solves the requirements of all users is still not possible. There is no single "customer" to define the requirements, instead they must be gathered from the community and from the users. This is an enormous task, and there is no guarantee that all the right users will be heard, and new requirements and use-cases will most likely appear in the future. Traditional requirement-driven software development processes that expects the requirement to be well defined early in the process, are not well suited for this situation.

Realizing this, the purpose of creating "DARC" is not to fulfill all requirements at once. Instead, it is a showcase and a prototype of a fundamentally different architecture for a control middleware based on the following principles:

- Only fundamental requirements are locked early in the process. These are defined as the requirements that have a fundamental impact on the core architecture.

- Non-fundamental requirements are implemented as customizable and loosely coupled. Thus the architecture is prepared to handle newly discovered requirements and use-cases.

6.2.1 Fundamental Requirements

Transparent Distribution

One of the powers of ROS is that nodes can be freely placed on different hosts on the network, as long as the ROS network protocols are capable of establishing a connection. This fundamental requirement is therefore inherited from ROS.

Decentralized

Unfortunately, the topology in a ROS system is greatly constrained by the need to have one central authority, the ROS master running on one of the host. Due to this, the topology of nodes must grow from this central point. It is difficult to connect two ROS control systems that have been started separately, e.g. for multi-robot cases. A new fundamental requirement is that the need for this central authority must be eliminated. Instead the cooperation must be handled in a decentralized manner.

Multi Linguistic

A fundamental design choice in ROS was that it should be able to program nodes in different programming languages, and these nodes should be able to communicate. A great power of ROS is that it now support several different programming languages such as C++, Python, Java and Lisp. This fundamental requirement is therefore inherited from ROS.

Reliable and Predictable Behavior

The reliability requirement can be a little difficult to relate to for average users. ROS is built for research, thus it is often enough to just get a system up and running during an experiment or for a demonstration. While ROS systems can appear to run fine for long periods, the system is designed in such a way, that some errors are silent and allows the system to continue running in an unpredicable manner.

Reliability is the ability of the system to perform its function even during unexpected circumstances. Clearly, some hostile circumstances such as network failures, program crashes or CPU congestion will prevent the system from functioning. But in these cases it is important that the error handling behavior is predicable, and the affected parts of the system is prevented from running. This requirement is categorized as fundamental because it is essential for building industrial and consumer products and it affects all layers of the architecture.

Support for Low Power and Embedded Systems

Because ROS was originally designed for the Willow Garage PR2 robot which includes two high performance computer systems, it has never really been designed to run on low performance systems. To properly support these embedded systems the middleware should use minimal resources and cause minimal performance overhead. And the same should apply for the dependencies it brings in. Therefore it is classified as a fundamental requirement.

6.3 A Multi-paradigm Middleware

The first attempt to design a ROS middleware replacement resulted in a prototype implementation in C++, where user functionality was implemented in components that could be either local or distributed over network. Components exchanged data using the ROS message format through an high performance asynchronous publisher & subscriber mechanism, and bookkeeping was handled in a decentralized manner. Therefore the name DARC was given, meaning "Decentralized Asynchronous Reactive Components".

From this version the middleware has evolved into much more, based on initial feedback, and a extensive analysis of the requirements. The core concept of performance and decentralized nature remains. The newest version includes:

- Decentralized peer-to-peer network, thus no central authority required.
- Full transparency between fast intraprocess communication, and network communication.
- Easy to combine multiple control systems, e.g. for multi-robot situations.
- Non intrusive supervision of the running system.
- No constraints on what data-types to use.
- Pluggable features.

The middleware is designed to be multi-paradigm, meaning that it can be used in many different ways.

The component model is only one possible paradigm, that is intended to be used for the customizable and mature parts of the control system, being high performance and intended to be implemented in C++. Components are very powerful, their lifetime can be managed and they can potentially be configured with a model driven approach.

Another option is the scripting paradigm, intended to be prototype implementations in a high level language like Python. The lifetime is controlled manually by running and stopping the script, and performance is less important.

A third paradigm is the source or sink paradigm, for exposing data streams from sensors or receiving commands for actuators. These parts of the system are independent from the actual control system. They don’t need lifetime management, as they can ideally be running all the time and be available for any control system that connects to them.

These are just the three current paradigms. Web interfaces, simulators and miscellaneous user interfaces will probably be optimally designed with other paradigms.

Since the bookkeeping is decentralized, control systems with different paradigms and different lifetimes can be dynamically formed and torn

apart again. At the lowest level they all speak the same platform independent protocol, so everything is DARC-native, there is no need for bridges. Robots and sensors can potentially run their own local independent DARC-system, thus exposing their data and receive commands in a native way.

6.4 Design of DARC

This section is not intended to be a complete and exhaustive design specification. It is intended to explain how DARC is designed to include these novel features and why it is claimed to be a next-generation middleware.

6.4.1 Programming Language

High flexibility, extendability and performance is obtained by creating the reference implementation in Modern C++ using the C++03 standard. Modern C++ is a style of C++, where many error-prone features inherited from C is avoided. Instead it utilizes a high use of standard library algorithms, smart pointers and programming techniques based on C++ templates.

Modern C++ can put some constraints on the compiler, but in practice only relatively simple template patterns are used, so most recent versions of the popular compilers, also for embedded systems, are capable of handling it. Template programming techniques, such as the policy pattern[1] is used to inject customizations of central functionality either at compile time or at runtime. This makes it possible to choose and override serialization method, memory allocation strategy, and even add new communications paradigms. The default choices are the most user-friendly suitable for standard users, but expert users are not constrained to these choices.

Support for multiple languages can be provided in two ways. (1)

[1] http://en.wikipedia.org/wiki/Policy-based_design

By using a wrapper library such as Swig[2] or Boost Python[3] to wrap the reference C++ implementation by a higher level language. (2) By implementing DARC, or a subset of DARC, natively in the respective language using the language independent protocol. The protocol is based on MessagePack[4], which is a high performance library for data exchange with support for a basically any modern programming language.

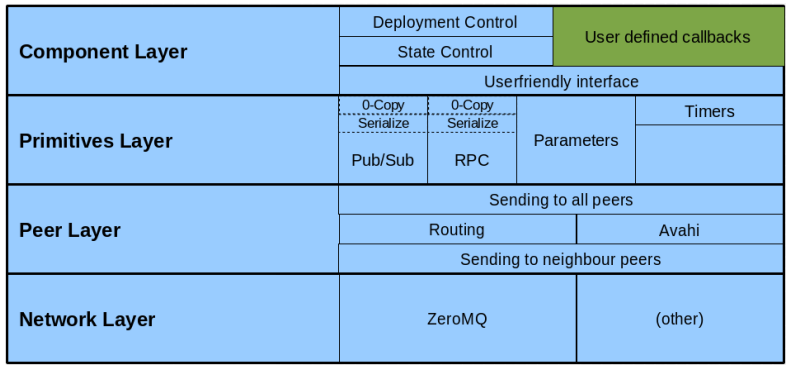


Figure 6.1: Architecture of DARC C++ Reference Implementation

6.4.2 Architecture

The architecture of the reference implementation is designed very strictly to support the required customization and extensions. As illustrated in figure 6.1 it is built up by four layers.

Network Layer

The transport protocol used for network distribution is abstracted by the network layer. The reference implementation uses the TCP transport from the ZeroMQ[5] network library. It is chosen because it provides platform independent socket connections, and among other things, adds

[2] http://www.swig.org
[3] http://www.boost.org/libs/python
[4] http://msgpack.org
[5] http://www.zeromq.org

guaranteed delivery of large messages and automatic reconnection. Because the transport is separated from the higher layers, new protocols such as broadcasted UDP, encrypted TCP or even RS232 serial links can be added with no changes in the higher layers required.

Peer Layer

The peer layer implements a routable network that abstracts the underlying network connections, described in more details in section 6.4.3 It implements supporting functionality for handling distributed records that higher layers can take advantage on.

Primitives Layer

The primitives layer implements the higher level communication patterns for data exchange, timer events and configuration, covered in more details in section 6.4.5. Because they are well separated in this layer it is easy to extend the middleware with new communication patterns if new use-cases arise. Primitives uses zero-copy methods for exchanging data when running on the same peer, while data to primitives located on other peers is serialized and routed through the peer layer. In theory, some communication patterns can also be constrained to run only on the same peer, e.g. to support real-time constraints.

Component Layer / Interface Layer

While the primitives layer can be used directly by a user application, it is often more advantageous to provide a more natural and user-friendly interface. In the C++ reference implementation the component layer provides a high performance, component based approach to implement user functionality. The interface follows an easy to use and C++ natural programming style, designed for non-experts, and hides low level functionality such as threading and deployment.

When wrapping the C++ reference implementation e.g. in Python, the component layer can be replaced by a different interface layer that

better supports the wrapper in providing a Python friendly programming style. It has also been experimented with implementing a ROS-node interface layer that is source compatible with ROS, so nodes written for ROS can be directly compiled to instead use the DARC middleware.

6.4.3 Peers

DARC is designed to run as a decentralized peer-to-peer system, where each peer runs as a separate process. The peers can be distributed on different hosts, or even implemented in different programming languages. Peers must be connected to form a DARC system. This can either be done manually, or the peers can be configured to discover each other automatically using the Zeroconf protocol.

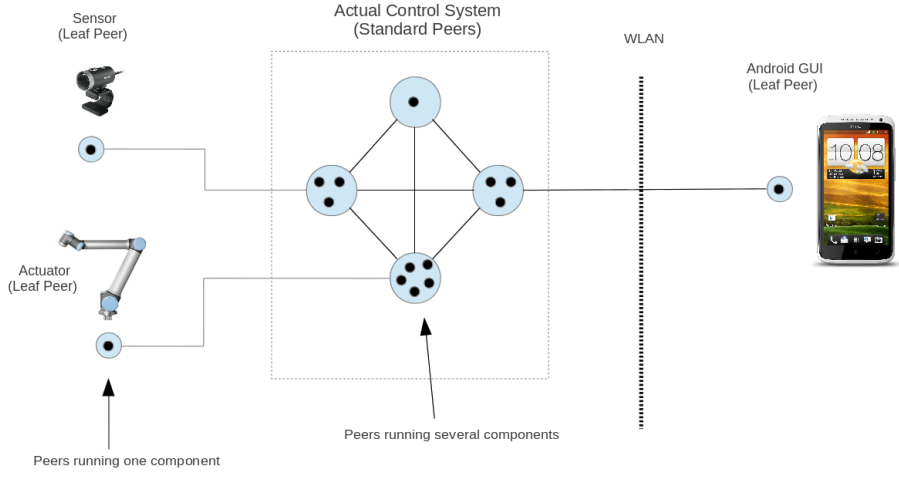


Figure 6.2: Peer Topology

A group of connected peers form a routing capable network, thus a peer needs only connect to one other peer and it is part of the control system. This makes it easy to connect GUI's, or combine two existing DARC systems even through wireless networks or Internet tunnels, as it can be done with a single connection.

The architecture is designed to be scalable in complexity. This makes

it possible to create a category of peers called “leaf” peers. “Leaf” peers can only be connected to one other peer and lack certain features such as routing and internal shared memory transport. As such they can only exist as leafs in the network topology. This type of peer works well for components only receiving sensor data, translating commands to actuators or GUI applications. Since they only need to implement a subpart of the functionality they are easier to port to other languages (e.g. Java for Android applications). Figure 6.2 shows the topology of an example DARC system where the main control system is implemented with normal peers and sensor data and GUI is handled by “leaf” peers.

6.4.4 Components

The user builds a control system by implementing her algorithms and functionality in DARC components, or by utilizing existing component implementations.

The component model is effective to decouple the different subparts of the control system into reusable building blocks. High reusablity is important since: (1) It minimizes the efforts required to create a control system for a new robot. (2) Reused components tends to be more mature, robust and throughly tested.

Components are compiled to a dynamic library and loaded into a peer. From here they will be able to communicate with all other components in the system, transparent whether they are running on the same peer or on a remote peer. The framework takes care of loading and instantiating the components and lower level functionality such as threading, error handling and logging. The robustness of the control system is improved by having the system state of all components being controlled by the state machine in figure 6.3. The component will not start execution until all prerequisites are available, such as remote resources, data ports are connected or parameters are loaded.

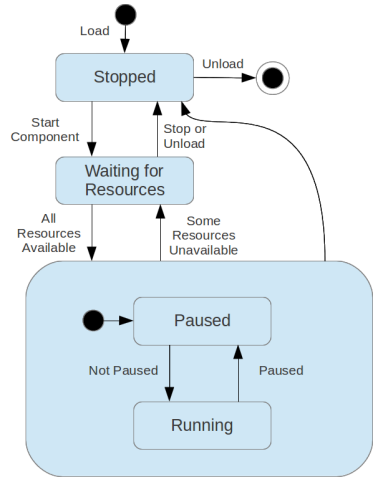


Figure 6.3: Internal Component State-Machine

6.4.5 Primitives

Components use a range of primitives to exchange data, synchronize with each other, and for configuration. The following types are available:

- Publishers & Subscribers
- Procedures
- Functors
- Timers
- Passive Resources
- Parameters

The user implements callback functions triggered by the primitives in case of events. Figure 6.4 shows the composition of a component.

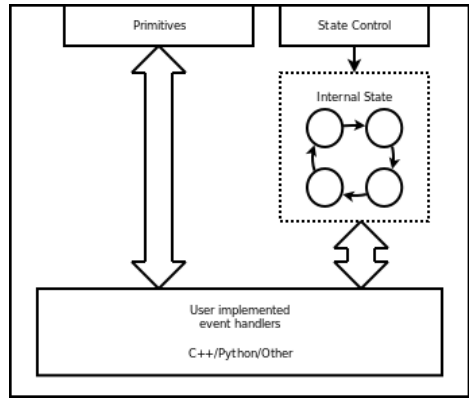


Figure 6.4: Component Model

Publishers & Subscribers

Components can publish data to a topic, making it directly available to any component subscribing to the same topic. This provides a powerful communication model for both periodic data streams, or for state information that should be globally available. At design time the publisher do not need to know about the existence of subscribers. By default a publisher is non-unique, allowing several components to publish to the same topic, implementing a many-to-many communication model. A publisher can also be configured to be unique, allowing only one component to publish to the topic, in situations where several sources is a design violation.

Procedures

DARC procedures are implemented using two types of primitives, a procedure server and a procedure client. Together they provide a request/reply communication model, useful for both:

1. Asynchronous requests for data which is expensive or otherwise unsuitable to publish, e.g. map data
2. Controlling and supervising long running pre-emptible tasks, e.g. commands to move a robot arm.

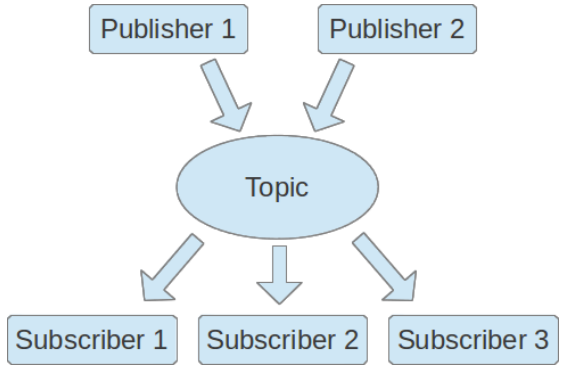


Figure 6.5: Publisher & Subscribe Pattern

At completion, the server returns a message with the result. In case of a longer running task, the server can return feedback messages with intermediate results. The task can be restarted or stopped by the client while it is running. Figure 6.6 shows an example sequence of a successful procedure call.

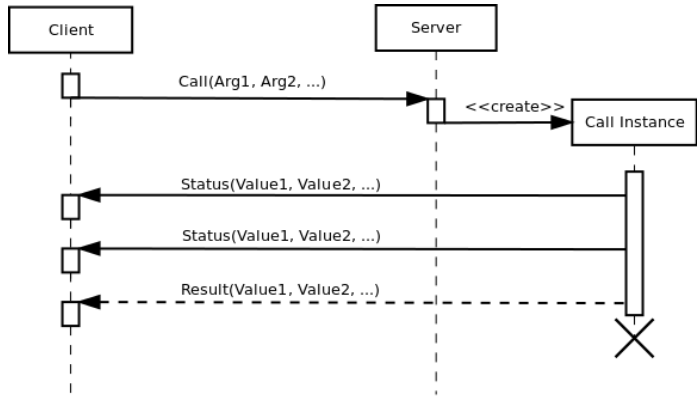


Figure 6.6: Procedure Call Sequence

Functors

Functors provide a mechanism to perform a transformation to a list of data of the same type, resulting in a list of result values. This model

fits the case where the calling component has the source data and also needs the result, but where the actual transformation is provided by an external component.

In existing frameworks this model is difficult to implement optimally. It is often emulated using publisher/subscribe or request/response models adding unnecessary performance loss and complexity. The functor model can improve the separation of concerns in a control system significantly.

$$\mathbf{y} = f_{\{\mathbf{p}\}}(\mathbf{x}) \tag{6.1}$$

$$\begin{bmatrix} \mathbf{y_0} \\ \mathbf{y_1} \\ .. \\ \mathbf{y_n} \end{bmatrix} = f_{\{\mathbf{p}\}} \left(\begin{bmatrix} \mathbf{x_0} \\ \mathbf{x_1} \\ .. \\ \mathbf{x_n} \end{bmatrix} \right) \tag{6.2}$$

A functor must fit the model in (6.1) and (6.2), where $\mathbf{x}$ and $\mathbf{y}$ contains vectorized data with the same data types respectively. The transformation f is parameterized with the parameters $\mathbf{p}$, and should be purely functional thus:

1. It should have no observable side effects.

2. It should yield the same result $\mathbf{y}$ for the same values of $\mathbf{x}$ and $\mathbf{p}$.

Using a functor is a two step process. (1) Providing the parameters $\mathbf{p}$ and acquiring a functor instance for $f_{\mathbf{p}}$. (2) Subsequent calls to the functor object with values of $\mathbf{y}$ as argument.

Functors are suitable for problems such a collision checking a set of poses, calculating state transitions e.g. odometry, or provide an abstract representation of a distribution that can be used to draw samples. Compared to procedures, functor calls are designed to be faster and easier to parallelize.

Timers

As DARC components are purely event based, reoccurring or delayed events should be triggered with timers instead of using loops or wait statements. Two types of timers exists. A periodic timers triggers the event handler with a constant interval, for reoccurring events. A deadline timer triggers the event handler once after it has been started, for delaying events or providing timeout supervision.

Passive Resources

Passive resources are purely used for synchronizing components and modeling higher level dependencies between components. A component dependent on a certain passive resource will not start before another component is loaded providing the passive resource.

Parameters

For maximum reusability, any adjustable variable should be created as a parameter. The actual parameter values are configured at load time, or can be adjusted while the system is running. DARC is designed to implement the “Smart Parameter Framework”, described in chapter 7, which makes it possible to derive the actual parameter values based on a robot model.

6.4.6 Data types

DARC supports, but is not limited to, the use of ROS data-types. Any copyable C++ data can be transported between components by the primitives. Large data can be passed using reference counted shared pointers, resulting in zero copying of data if the components are loaded into the same peer. To pass data between components located on network separated peers, the data structure must be serializable. DARC can use any serializer method, such as ROS serializer, boost serializer or a user defined method.

6.5 Real-time Support

Some parts of the control loops must run at high frequency, with a completely guaranteed execution time, known as hard real-time. DARC is not intended to support hard real-time execution, instead it is encouraged to implement those subparts in a framework specially designed for this, such as The Orocos Real-time Toolkit[6]. Instead DARC focuses on the part of the control system which can run in soft real-time. Soft real-time means execution and deadlines are allowed to have a certain slack, and DARC provides tools to supervise that this slack is fulfilled.

6.6 Performance Test

Using frameworks such as DARC and ROS provide powerful ways to split functionality into reusable subpart. But they will always result in some amount of performance loss compared to raw C++ function calls. Minimizing the performance loss is important since it allows the system to be split into smaller parts.

Comparing the latency performance of DARC and ROS is done using two test cases where two components(DARC case), or two nodes(ROS case) communicate. In the DARC case the components are either loaded into the same peer to make use of the intra-process communication optimizations, or into two peers located on the same host but connected through TCP. In the ROS case the nodes are also located on the same host and connected through TCP. Using Nodelets[7] in ROS does allow for zero-copy transport and lower latency. But while DARC components supports both patterns out of the box, ROS Nodelets requires the node to be specially implemented and thus not available to the average user.

Two test systems were used:

1. Beagleboard-xM with a 1GHz Arm® Cortex™-A8 CPU. Running Ångström Linux.

[6] http://www.orocos.org/rtt
[7] http://www.ros.org/wiki/nodelet

2. Laptop equipped with a 2.40GHz Intel® Core™ i5 CPU. Running Ubuntu 12.10.

The native supported system by ROS is Ubuntu, and compiling and running the ROS nodes on Ångström Linux, at the time the tests were run, turned out to be quite a challenge. Therefore all the DARC/ROS comparison tests were performed on the Ubuntu system.

Instead only the Publisher & Subscribe test of DARC was run in the Beagleboard to show the performance of DARC in an embedded system.

6.6.1 Publish & Subscribe

The speed of the publish & subscribe pattern is tested by “Component A” publishing a small ping message received by “Component B”. Upon receiving the ping, “Component B” publishes a small pong message received by “Component A”. The average round trip time of the ping/pong communication is measured over a period of several seconds. See table 6.1 and table 6.2 for results.

	Same Peer/Node	Distributed Peer/Node
DARC	$\sim 1.0\mu s$	$\sim 50\mu s$
ROS	N/A	$\sim 180\mu s$

Table 6.1: Speed of DARC and ROS publisher/subscribe. (Ubuntu System)

	Same Peer/Node	Distributed Peer/Node
DARC	$\sim 30\mu s$	$\sim 4ms$

Table 6.2: Speed of DARC publisher/subscribe. (Beagleboard-xM Test System)

6.6.2 Procedures & Actions

“Component A” performs a call to the procedure or action provided by “Component B” and waits for the result. Again the average round trip

time of the request is measured over a period of several seconds. See table 6.3 for results.

	Same Peer/Node	Distributed Peer/Node
DARC	$1.1\mu s$	$\sim 55\mu s$
ROS	N/A	$\sim 1600\mu s$

Table 6.3: Speed of DARC procedures and ROS actions. (Ubuntu Test System)

6.7 Conclusion

The design of DARC shows that it has been possible to improve several areas of state-of-the-art robotics frameworks. Both concerning performance and support for new use-cases. The component model improves the ability to create reusable software. By exchanging data using DARC communication primitives, it is transparent at design time whether components are running on the same host or on network separated hosts. The functor primitive further improves the ability to decouple the components. Several tests show an enormous latency improvement, due to DARC components running in the same peer utilizing zero copy data transfer.

7

Smart Parameter Framework

7.1 Introduction

This chapter presents the "Smart Parameter Framework" that can be used for configuring the individual components in component based robot control systems. Using smart parameters that adapt to the respective robot system makes it possible to obtain optimal parameter values while reusing the software components, without expert knowledge about the underlying algorithms. The framework derives algorithm specific parameters from more high level and understandable robot properties, which can often be measured or calibrated. Therefore the framework also assists in building robot systems that can autonomously calibrate itself, resulting in higher stability of the robot and less tuning required.

The framework was initially designed to be general and usable in several different robot middlewares, while the intended purpose has later shifted towards being the core parameter system for the DARC middleware. This chapter will describe the initial generic approach.

7.1.1 Problem Formulation

An autonomous robot system consists of many different technologies for sensing and control, and as a result each system is often unique to some extent. This requires the control software and control algorithms to be designed or adapted for each particular robot system, leading to important resources wasted on reinventing the wheel. Designing and integrating these technologies can be complex, resource demanding and requires specialized knowledge. This which hinders the full potential for developing commercial robot applications where development resources and project risks should be kept at a minimum. Fortunately, much focus has been put on creating robot control frameworks with modularity and reusability in mind, using design patterns originating from traditional software design such as active objects and data-flow patterns[Bru01][Ger+01][Qui+09].

Using such a framework can release the developer from much implementation work, but it still requires a considerable effort in integrating the components. The task includes the complex job of configuring each component, some of which contains advanced algorithms and control. Traditionally, this configuring is performed with primitive methods such as hardcoding constant values defined in the source code, or supplying parameter values during deployment or runtime. Since each of these parameters must be adjusted to the specific robotic application, it again requires deep insight to the underlying algorithms and also adds a potential source of errors. Figure 7.1 shows a piece of the source code for the AMCL localizer node[1] in ROS, where parameters are defined in the source code. The meaning of the parameters are difficult to understand without knowing the AMCL algorithm. Ideally, the configuration should be performed by the specialist who has designed and implemented the component. In practice, this is not possible when using primitive parameters, since the component designer can only set a default parameter value and possibly an exhaustive description of each. It is still the responsibility of the integrator to verify that the parameter values are correct and

[1] http://www.ros.org/wiki/amcl

optimal for the respective robot application.

```
private_nh_.param("odom_alpha1", alpha1_, 0.2);
private_nh_.param("odom_alpha2", alpha2_, 0.2);
private_nh_.param("odom_alpha3", alpha3_, 0.2);
private_nh_.param("odom_alpha4", alpha4_, 0.2);
private_nh_.param("odom_alpha5", alpha5_, 0.2);
private_nh_.param("laser_z_hit", z_hit_, 0.95);
private_nh_.param("laser_z_short", z_short_, 0.1);
private_nh_.param("laser_z_max", z_max_, 0.05);
private_nh_.param("laser_z_rand", z_rand_, 0.05);
private_nh_.param("laser_sigma_hit", sigma_hit_, 0.2);
private_nh_.param("laser_lambda_short", lambda_short_, 0.1);
private_nh_.param("laser_likelihood_max_dist",
                  laser_likelihood_max_dist_, 2.0);
```

Figure 7.1: C++ source code defining the primitive and difficult to understand parameters for the AMCL localizer in the ROS Navigation Stack. **Source:** `https://github.com/ros-planning/navigation`

Some properties of the robot or the environment might be expected to change over time, or are unknown at design time. In such case it is not possible for the integrator to fully configure the robot using only primitive parameter values. Instead they must be calibrated by the robot itself using methods such as [Lar+98][RT98][SH07][SS04]. Integrating extensive self calibration procedures in a complex robot system is not straightforward. The robot must often be allowed to perform certain actions to perform the calibration, such as attempting to move or activating other actuators. With unknown properties the result of the actions are uncertain, and safety measures such as collision avoidance and compliance to dynamic constraints can not be fully guaranteed. Instead the robot must perform these actions with slow speed or other conservative safety measures. Therefore the system must have access to meta information about when some parameters are unknown or uncertain, and take the proper precautions.

7.2 Smart Parameters

This article proposes the "smart" parameters concept where the parameters are not configured to have a constant value, but instead described as relationships and dependencies to other properties or parameters of the robot system. This allows the component developer to perform meta configuring of each component, by configuring the relations and requirements for each of the component parameters. The burden of configuring can thus be handled in advance by the robotic expert who knows the algorithms in detail, and the integration is performed by supplying a descriptive model of the properties of the robot system. The components can in most cases be easily and safely reused by a non-robotics expert, without the need to adapt it manually.

The properties and parameters in the system includes meta information about the source of the values. A property can thus be specified as unknown, guessed, measured or calibrated, and the parameter values derived from the respective property will be inherit the source information. The control system can use this information to enter certain safe modes or initiate the required calibration procedures. Due to the smart parameters, each dependent component parameter will be updated based on the calibrated properties.

This gives a configuration framework for fast prototyping and autonomous calibrating robots, and allows for supervision of critical parameters in the robot system. The purpose of the framework is to improve the development process and allow for more competitive and robust robot applications to be developed for the commercial market.

7.3 Architecture

The architecture for supporting smart parameters is designed as an extension to a component-based robotics framework such as OROCOS[Bru01] or ROS[Qui+09]. It consists of four main parts: The descriptive models, the parameter server, the "dynamic parameter" software pattern and the

control GUI as illustrated in figure 7.2.

7.3.1 Descriptive Models

Two types of models are used to describe a system based on smart parameters.

The first type is the system models, which are the source of information about the actual robot and environment properties such as physical properties, mechanical configuration, available resources, sensors, etc. The models are designed to be combined in a hierarchical structure to easily integrate existing models for often used hardware and sensors. Similar descriptive models, such as world files in the Player/Stage[Ger+01] framework and URDF files in ROS[Qui+09], exists.

In addition, each software component must be accompanied by one or more component models, describing the full list of parameters the component expects, and how they should be derived. Using python as scripting language, and bindings to access properties in the robot model and other component models, makes it possible to set up complex rules to derive the respective values for each parameter. Several prioritized rules can exists for each parameter. The highest priority rules will be tried first and skipped if any of the properties the rule depends on is unavailable. The lowest priority 'default' rule is to flag the parameter as unknown. Allowing a component to start with an unknown parameter value might be useful in certain situations e.g. if the value is expected to be autonomously calibrated. Similar to object oriented programming, the component models can inherit from each other and only extend or override the rules required for the specific implementation. This makes it possible to maintain several component models intended for different purposes while keeping a single component implementation.

7.3.2 Parameter Server

The core of the architecture is the parameter server, which exists as an individual component in the robot system. The parameter server is

responsible for parsing the rules configured in the component models and dispatching the derived parameter values to the correct components, both when the system is loaded, and when parameter changes are required. The parameter server exists as a separate software library, and is not dependent on the actual middleware used to create the robot control system. It only have to be wrapped to uses the communication model of the respective middleware to dispatch the parameter values to each of the active software components in the system.

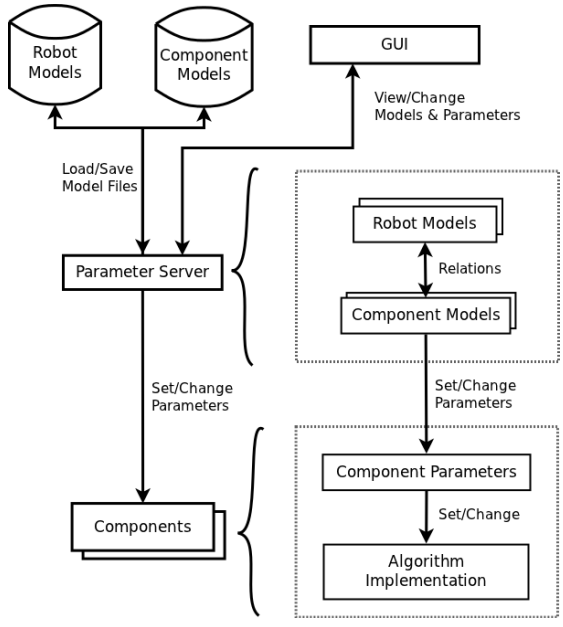


Figure 7.2: Architecture Overview

7.3.3 Dynamic Parameter Software Pattern

The implementation of each algorithm or software component in the system must be designed to read the correct parameter values from the parameter server before starting, and to handle dynamic parameter updates properly. In practice this is supported by a series of existing software classes the component can extend. This just requires the component

to implement a number of callback methods to handle changes in the parameters. In addition, it adds a simple state control of the algorithm to start only when valid parameters have been loaded, and to restart the algorithm if the parameter updates require such an action.

7.4 Configuration Example

The robotics lab at the Technical University of Denmark includes several mobile robots of varying size, and sensor configurations. Whenever a new robot is adapted, or an existing robot configuration is changed, the corresponding robot control system must be reconfigured accordingly. The most basic configuration required is to define the actual physical properties of the wheel base such as the size and position of the wheels. Trivial as this task might seem, it is important to perform properly to make the control system perform optimally.

The smart parameters framework has been used to configure a subpart of the control system for the differential drive mobile robot shown in figure 7.3. The subpart includes the software component calculating odometry based on encoder data from the wheels, a component for controlling the wheel speed, and the localization system.

Figure 7.3: The smart parameters framework has been used to configure a subpart of the control system for a ATR-JR mobile robot

The setup is very simple and rather straightforward to configure for the specific case, but it serves as a basic example for examine the potential of the smart parameter framework. Many of the parameters for the system are related to the geometric and dynamic properties of the robot, and require adjustments if a different robot is used.

The system and component models required to configure the system are illustrated in figure 7.4.

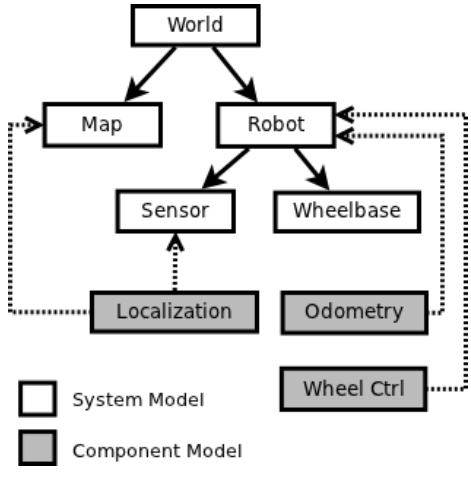


Figure 7.4: Relations between the system and component models used to configure the example system

7.4.1 System Models

In the example the 'world' model represents the root, which has a reference to one robot and a map model. The laser-scanner on the robot is described in the 'sensor' model and the wheel dimensions in the 'wheelbase' model. This model tree represents the physical robot and the environment, and includes all the required properties for the example system. Each component model is associated with one or more system models, which are used for searching the tree for the required information. Bottom-up search is used so that the nearest ancestor is always related to the respective robot and the search will work also for incom-

plete trees. If other robots exists in the model, they can be referenced through the world model.

```
# dtu_atrv.sm
# System model file for DTU ATRV-JR robot
---
name: dtu_atrv
parent: local://models/mobile_robot.sm
size:
  parent: system://boundingbox_3d
  source: measured
  value: { x: [-0.46, 0.56],
           y: [-0.33, 0.33],
           z: [ 0.00, 0.60] }
sensor1:
  parent: local://models/sick-lms100.sm
  pose:
    parent: system://pose
    position:
      source: measured
      value: { x: 0.43, y: 0, z: 0.39 }
    orientation:
      source: measured
      value: { x: 0, y: 0, z: 0 }
wheelbase:
  parent: local://models/diff_drive_4_wheel.sm
  displacement:
    source: measured
    mean: 0.53
    variance: 0.02^2
  wheeldiameter:
    source: 'unknown'  # <-- wheel size unknown
```

Figure 7.5: System model file for the ATRV-JR robot used in the experiment. The file is created using YAML format, and specifies the physical size, the available sensors, and the properties of the wheelbase.

Figure 7.5 shows the system model for the ATRV-JR robot used in the experiment. The system model files are created using the extensive and human readable YAML syntax. The YAML format allows a hierarchical structure to be created. Any entry in the hierarchy can inherit and extend an existing robot model entry using the parent keyword. It can

access system build-in models using "system://", local model files using "local://" or model files through the network using a variety of protocols such as "http://".

Special focus should be put on the wheelbase section of the model. This section extends the "diff_drive_4_wheel" system model, thus inheriting some generic properties for a four wheel differential drive wheel base. The wheel-diameter is configured as unknown, since it is expected to be calibrated autonomously. The smart parameter framework will propagate this information to any parameters in the system that expects to use the wheel-diameter.

7.4.2 Wheel Control Component Model

A component model declares the requirements and parameters for a specific software component. It consists of constants, simple rules to derive the values from the system model, or even small python programs for handling advanced cases. Similar to the system models, the component model use YAML format and support inheritance from other component models. This allows several component models to exist for each software component, adjusted to handle different purposes.

Figure 7.6 shows a part of the component model configuring the software component controlling the wheels. The requirements section is designed to catch situations the respective software component is not designed to handle. E.g. a differential drive motor control component used for a omni-directional robot. If the requirement is not fulfilled, the component will not run and report to the system integrator that he has chosen an incorrect component for its use. The parameter section contains constants or rules to derive the values of the software component's parameters. The "update_time" parameter is set to a constant, while the wheel size and wheel displacement parameters are derived directly from the respective robot model. These will thus adapt to changes in the robot system model, including updates resulting from calibration.

The model also derives a value for the maximum safe velocity the

robot is allowed to move with. The controller can only map this maximum velocity to a maximum angular wheel velocity if the wheel-size is known. Before this value is calibrated, it must be excessively cautious. This situation is handled by configuring the "max_angular_velocity" parameter with a small python script located. If the wheel-diameter is not yet calibrated, a worst case value is calculated based on the mean and standard deviation of the current guess.

```
# dtu_wheel_ctrl.cm
# Component model for WheelCtrl component
---
name: dtu_wheel_ctrl
requirements:
 - condition: robot.wheelbase.type == 'differential'
parameters:
 - name: update_time
   type: float
   value: 0.1
 - name: displacement
   parent: system://gaussian
   derive: robot.wheelbase.displacement
 - name: wheeldiameter
   parent: system://gaussian
   derive: robot.wheelbase.wheeldiameter
 - name: max_safe_velocity
   type: float
   derive: robot.dynamics.max_safe_velocity
 - name: max_angular_velocity
   python: |
     if wheeldiamater.source != 'calibrated':
       value = max_safe_velocity /
               (wheeldiameter.mean +
                wheeldiameter.std_dev * 2)
     else:
       value = max_safe_velocity /
               wheeldiameter.mean

...
```

Figure 7.6: Part of the component model for the Wheel Ctrl component.

7.4.3 Odometry Component Model

The Odometry Component Model is associated with the model representing the robot in the system. It also requires the wheelbase of the respective robot to be of type differential drive as shown in figure 7.7.

```
# dtu_odometry.cm
# Component model for odometry component
---
name: dtu_odometry
requirements:
 - condition: robot.wheelbase.type == 'differential'
parameters:
 - name: update_time
   type: float
   value: 0.1
 - name: displacement
   parent: system://gaussian
   derive: robot.wheelbase.displacement
 - name: wheeldiameter
   parent: system://gaussian
   derive: robot.wheelbase.wheeldiameter
 - name: translational_error
   type: float
   python: |
     if wheeldiamater.source != 'calibrated':
       [value,source] = [0.5, 'guess']
     else:
       [value,source] = [0.05, 'guess']

...
```

Figure 7.7: Part of the component model for the Odometry component. The component requires a differential drive wheelbase, and derives a few parameters from the robot system model shown in figure 7.5.

The value for the wheel displacement, and wheel size of the robot is derived directly from the 'wheelbase' model. The value of `translational_error` is set to a large value as long as the wheel-size has not been calibrated. In both cases the value is guessed, and the meta-parameter `source` is set to 'guess' to clarify this.

7.5 Conclusion

Using 'smart' parameters allows the robotics developer to perform meta configuring of the software components she is implementing. Each software component is accompanied by a component model describing the requirements for the component to run and rules for deriving the optimal parameter values. This makes it possible to reuse the components without expert knowledge about the algorithms, by supplying a system model describing the properties of the robot. The components will automatically adapt to the new system.

Using meta information associated with each parameter allows the system to take actions due to unknown of imprecise parameters, such as running in safe mode or initialize calibration procedures.

This chapter presented a proof-of-concept system, where the 'smart' parameters were successfully used to configure a control system where a mobile robot should calibrate the size of the wheels and localize itself in a known map using a laser scanner. Reusing the control system on a different robot requires only changing the robot model, saving precious development time and guaranteeing that the components are configured optimally.

8

Conclusion

8.1 Conclusion

This research project was motivated by the overall goal of reducing the cost and risk associated with developing commercial applications based on mobile robots. The intended approach was to influence and contribute to the research community to make the resulting work and implementation more ready to be used for industrial and commercial products. During the project, this contribution was shifted towards influencing the ROS framework and the ROS community to focus on performance, quality and provide support for embedded system.

1. Generic Navigation

A fundamental problem for robot applications based on mobile wheeled robots is to be able to drive around and navigate the surrounding. Unfortunately no implementation of a generic and easy-to-use navigation solution was found. Having to implement the navigation for each new robot is not an optimal solution for either commercial or research applications. One of the scientific goals listed in chapter 1 was to mature one or more algorithms for improved commercial use. Because a robust and generic navigation solution is such a fundamental requirement, focus was

put on maturing this part.

During this project, a part of such a generic navigation solution was designed and implemented, with focus on differential drive and Ackermann robots. The geometrical and dynamic constraints of these types of robots were formalized and represented by a generic constraint function for calculating the maximum length of the robot velocity vector at a given curvature.

The work included a generic representation for curved trajectories with both position and orientation information, capable of representing rotation-on-the-spot and reverse motion. Methods for making such a trajectory drivable with continuous curvature, and for calculating a feasible and safe velocity profile based on the constraint function, was presented and verified with two example trajectories.

In addition, a trajectory-following controller capable of handling both rotation-on-the-spot and reverse motion was designed. The controller was verified with a practical experiment using a differential drive robot.

The experiments showed that the approach was capable of both representing and following a trajectory. A few problems was identified, namely that the trajectory would be slightly altered when creating continuous curvature, potentially resulting in too high curvature or a trajectory with collision. A solution to this problem could be to allow for moving the knot points while considering collision situations.

2. Robot Middleware

During the project, a compatible alternative to the ROS middleware called DARC was designed and implemented. The design of DARC was a result of a exhaustive collection of lessons learned from using ROS, and newly identified requirements and use-cases resulting from industrial stakeholders. It was argued why DARC is branded as a next generation middleware. It is designed to be fully decentralized, scalable between small low power hardware, and large complex control systems. Performance tests showed how data-exchange using DARC was magnitudes

faster than the corresponding communication model in ROS

DARC now exists as a prototype implementation, and because it is compatible with the ROS data-types, much of the existing ROS functionality can be easily used. Because it is only a prototype, neither the stability of the API, nor the documentation has been mature enough to release it fully into the ROS community yet.

The development goal listed in chapter 1 was to contribute and influence the research community towards building higher quality software, and to provide tools for supporting this. DARC has has been a success in providing this influence, since the next version of ROS branded ROS 2.0 is highly inspired by the DARC prototype. It is planned to integrate the component model, the high performance data-transport, the highly modular architecture, and native support for embedded systems among other things. The decentralized nature is also considered.

In addition, several companies and ROS user-groups have shown a substantial interest in DARC, because is solves many of their problems in ROS concerning performance, reliability and ease-of-use. Therefore DARC has also potential to becoming an alternative to the ROS middleware. Because DARC has very little requirement for backwards compatibility with respect to old ROS functionality, it is much less-constrained in the design and much easier to implement new powerful features.

3. Smart Parameters

During the project, a paradigm shift in how parameters are used to configure component based control systems, was proposed. By using smart parameters, that adapts to the respective robot based on a model description, it is possible to reuse software components without expert knowledge about the underlying algorithms. A generic method and architecture to integrate this type of parameters into a robot control system was presented. The method was used to configure a differential drive robot based on a series of models.

One of the scientific goals listed in chapter 1, was related to using

parameterized models for describing the physical properties and behavior of a robot system, with the purpose of building self-calibrating and self-optimizing robots. The Smart Parameter framework shows a potential for solving this problem, but is still missing some efforts before it has been throughly tested, matured and used for building self-calibrating robots.

8.2 Future Work

During the project, the three areas of contribution listed above has been run separately. Future work is intended to properly integrate these three areas.

The high interest the DARC middleware has received from both industry and research, shows a large potential for continuing the work on maturing and documenting the prototype implementation. Future work is planned to investigate these possibilities further, and decide whether DARC should be developed as a high-performance and robust alternative to ROS, or whether the effort should be put into ROS 2.0.

Concerning the Smart Parameter Framework, the generic method is planned to be replaced by a native integration into the DARC middleware. From this, DARC would implement the model based configuration and smart parameters as the default configuration paradigm.

Only a part of a generic navigation solution was implemented during the project. Future work is required for adding trajectory planning, higher level obstacle avoidance based on inevitable collision states, and a more throughly tested controller for trajectory-following. Designing a controller that takes the respective robot models further into account, will make it possible to implement observers for detecting errors in the configured robot parameters. These observer values, together with the Smart Parameter Framework, will make self-calibration and self-optimization of the wheel configuration possible.

Bibliography

[AFF04] Hajime Asama, Thierry Fraichard, and Thierry Fraichard. "Inevitable collision states - a step towards safer robots". In: *Advanced Robotics* 18 (2004), pp. 1001–1024.

[Asu13] Asus®. *Asus XTion PRO product website*. Jan. 2013. URL: `http://www.asus.com/Multimedia/Xtion_PRO`.

[Bru01] H. Bruyninckx. "Open robot control software: the OROCOS project". In: *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*. Vol. 3. 2001, 2523 –2528 vol.3. DOI: `10.1109/ROBOT.2001.933002`.

[CAG08] Ricardo Carona, A. Pedro Aguiar, and José Gaspar. "Control of Unicycle Type Robots - Tracking, Path Following and Point Stabilization". In: *Proceedings of IV Jornadas de Engenharia Electrotécnica e de Computadores*. 2008.

[Cha82] R. Chatila. "Path planning and environment learning in a mobile robot system". In: *Proceedings of ECAZ, Orsay, France*. 1982.

[CM05] Toby H. J. Collett and Bruce A. Macdonald. "Player 2.0: Toward a practical robot programming framework". In: *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*. 2005.

[Cou92] R. Craig Coulter. *Implementation of the Pure Pursuit Path Tracking Algorithm*. Tech. rep. CMU-RI-TR-92-01. Pittsburgh, PA: Robotics Institute, 1992.

[Cro85] J. L. Crowley. "Navigation for an Intelligent Mobile Robot". In: *IEEE Journal on Robotics and Automation (Now known as IEEE Transaction on Robotics and Automation)* (1985).

[DJ00] Gregory Dudek and Michael Jenkin. *Computational principles of mobile robotics*. New York, NY, USA: Cambridge University Press, 2000. ISBN: 0-521-56876-5.

[FBT97] D. Fox, W. Burgard, and S. Thrun. "The Dynamic Window Approach to Collision Avoidance". In: *IEEERobotics & Automation Magazine* 4.1 (1997).

[Ger+01] B.P. Gerkey et al. "Most valuable player: a robot device server for distributed control". In: *Proceedings. 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. (IROS)*. Vol. 3. 2001, 1226 –1231 vol.3. DOI: 10.1109/IROS.2001.977150.

[GK08] Shilpa Gulati and Benjamin Kuipers. "High Performance Control for Graceful Motion of an Intelligent Wheelchair". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2008.

[GSB05] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2005.

[GSB06] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters". In: *IEEE Transactions on Robotics* (2006).

[Hae+03] D. Haehnel et al. “A highly efficient FastSLAM algorithm for generating cyclic maps of large-scale environments from raw laser range measurements”. In: *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*. 2003.

[HK07] Thomas M. Howard and Alonzo Kelly. “Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots”. In: *The International Journal of Robotics Research* 26.2 (Feb. 2007), pp. 141–166. ISSN: 0278-3649. DOI: 10.1177/0278364906075328.

[iRo13] iRobot®. *iRobot Ava Mobile Robotics Platform product website*. Jan. 2013. URL: http://www.irobot.com/ava.

[Kha86] O Khatib. “Real-time obstacle avoidance for manipulators and mobile robots”. In: *The International Journal of Robotics Research* 5.1 (Apr. 1986), pp. 90–98. ISSN: 0278-3649. DOI: 10.1177/027836498600500106.

[KS07] James Kramer and Matthias Scheutz. “Development environments for autonomous mobile robots: A survey”. In: *Autonomous Robots* 22 (2007), p. 132.

[Lar+98] Thomas Dall Larsen et al. “Location Estimation for an Autonomously Guided Vehicle using an Augmented Kalman Filter to Autocalibrate the Odometry”. In: *Proceeding of FUSION'98, LasVegas, Nevada, U.S.A.* 1998.

[Lik+05] Maxim Likhachev et al. “Anytime dynamic a*: An anytime, replanning algorithm”. In: *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*. 2005.

[LKJ01] Steven M. LaValle, James J. Kuffner, and Jr. *Randomized Kinodynamic Planning*. 2001.

[LL03] A. Pascoal L. Laperre D.Soetanto. “Non-singular Path-Following Control of a Unicycle in the Precense of Parametric Model-

ing Uncertainties". In: *International Journal of Robust and Nonlinear Control* (2003).

[LSB09] Boris Lau, Christoph Sprunk, and Wolfram Burgard. "Kinodynamic Motion Planning for Mobile Robots Using Splines". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2009)*. 2009.

[Mat13] Matterport. *Matterport 3D scanner product website*. Jan. 2013. URL: `http://matterport.com`.

[MF07] Christian M and El Udo Frese. "Comparison of Wheelchair User Interfaces for the Paralysed: Head-Joystick vs. Verbal Path Selection from an offered Route-Set". In: *Proceedings of the 3rd European Conference on Mobile Robots, EMCR 2007*. 2007.

[Mic+93] Alain Micaelli et al. *Trajectory Tracking for Unicycle-Type and Two-Steering-Wheels Mobile Robots*. Tech. rep. 1993.

[Mis13] Miscrosoft®. *Kinect product website*. Jan. 2013. URL: `http://www.xbox.com/da-DK/Kinect`.

[Mor80] Hans Moravec. "Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover". In: *tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University & doctoral dissertation, Stanford University*. 1980.

[QBN07] Morgan Quigley, Eric Berger, and Andrew Y. Ng. *STAIR: Hardware and Software Architecture*. 2007.

[Qui+09] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009.

[RT98] Nicholas Roy and Sebastian Thrun. "Online Self-Calibration For Mobile Robots". In: *Proceeding of the IEEE International Conference on Robotics and Automation*. IEEE Computer Society Press, 1998, pp. 2292–2297.

[Sah+07] A. Sahraei et al. “Artificial Intelligence and Human-Oriented Computing”. In: Springer, 2007. Chap. Real-Time Trajectory Generation for Mobile Robots.

[Sch87] M.J. Schoppers. “Universal Plans for Reactive Robots in Unpredictable Environments”. In: *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*. 1987.

[SG91] Zvi Shiller and Yu rwei Gwo. “Dynamic Motion Planning of Autonomous Vehicles”. In: *IEEE Transactions on Robotics and Automation* 7 (1991), pp. 241–249.

[SH07] Davide Scaramuzza and Ahad Harati. “Extrinsic Self Calibration of a Camera and a 3D Laser Range Finder from Natural Scenes”. In: *IEEE International Conference on Intelligent Robots and Systems (IROS 2007)*. 2007.

[SNS11] Roland Siegwart, Illah R. Nourbakhsh, and Davide Scaramuzza. *Introduction to Autonomous Mobile Robots*. 2nd. The MIT Press, 2011. ISBN: 0262015358, 9780262015356.

[SS04] Daniel Stronger and Peter Stone. “Simultaneous Calibration of Action and Sensor Models on a Mobile Robot”. In: *IEEE International Conference on Robotics and Automation*. 2004.

[VAP08] Francesco Vanni, A. Pedro Aguiar, and Antonio M. Pascoal. “Cooperative path-following of underactuated autonomous marine vehicles with logic-based communication”. In: *2nd IFAC Workshop Navigation, Guidance and Control of Underwater Vehicles 2nd IFAC Workshop Navigation*. 2008.

[Wyr+08] K. A. Wyrobek et al. “Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot”. In: *IEEE International Conference on Robotics and Automation, 2008. ICRA 2008.* 2008. DOI: 10.1109/ROBOT.2008.4543527.

www.elektro.dtu.dk
Department of Electrical Engineering
Automation and Control
Technical University of Denmark
Ørsteds Plads
Building 348
DK-2800 Kgs. Lyngby
Denmark
Tel: (+45) 45 25 38 00
Fax: (+45) 45 93 16 34
Email: info@elektro.dtu.dk

ISBN 978-87-92465-51-1